UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: **Gobstones y XGobstones: concretando lenguajes para enseñar a programar**
Autores: Ary Pablo Batista
Director: Gabriel Baum, Lic.
Codirector: Pablo E. Martínez López, Dr.
Carrera: Licenciatura en Informática

Resumen

A partir del diseño de lenguajes de programación es necesaria la implementación de herramientas que permitan el análisis y la ejecución de programas escritos en esos lenguajes.

Al considerar la enseñanza de la programación en personas con ningún conocimiento previo, en la Universidad Nacional de Quilmes abordaron el problema a través del diseño de una secuencia didáctica innovadora que dió forma a los lenguajes de programación Gobstones 3.0 y XGobstones.
En este trabajo se presenta el desarrollo de compiladores y máquinas virtuales para los lenguajes de programación Gobstones 3.0 y XGobstones y el diseño e implementación del mecanismo necesario para la interacción entre estos y la interfaz gráfica de PyGobstones 1.0, ambiente de desarrollo que incluye estas implementaciones.

Esta tesina tiene como resultado el desarrollo de las implementaciones propuestas, completando así la primer versión del ambiente de desarrollo PyGobstones 1.0 que actualmente es utilizado en la Universidad Nacional de Quilmes para enseñar las nociones básicas de programación.

Palabras Claves

– *Implementación de lenguajes de programación*
– *Compiladores*
– *Máquinas virtuales*
– *Análisis Sintáctico*
– *Módulo de interacción*
– *Ambiente de desarrollo*
– *Gobstones*
– *XGobstones*

Conclusiones

– *Se desarrolló un compilador y una máquina virtual que implementen el lenguaje de programación Gobstones 3.0.*
– *Se desarrolló un compilador y una máquina virtual que implementen el lenguaje de programación XGobstones.*
– *Se diseñó e implementó un módulo de interacción para que las implementaciones de los lenguajes Gobstones 3.0 y XGobstones puedan interactuar con la interfaz gráfica de PyGobstones 1.0.*
– *Estas implementaciones completan la primer versión del ambiente de desarrollo PyGobstones 1.0.*

Trabajos Realizados

– *Desarrollo de un compilador y una máquina virtual que implementen el lenguaje de programación Gobstones 3.0.*
– *Desarrollo de un compilador y una máquina virtual que implementen el lenguaje de programación XGobstones.*
– *Diseño e implementación de un módulo de interacción para que las implementaciones de los lenguajes Gobstones 3.0 y XGobstones puedan interactuar con la interfaz gráfica de PyGobstones 1.0.*

Trabajos Futuros

– *Desarrollar implementaciones de las máquinas virtuales que apunten a alcanzar un rendimiento comparable con el rendimiento de la máquina virtual de Python.*
– *Desarrollar un proceso de compilación optimizante que reduzca los tiempos de ejecución de los programas.*
– *Unificar las implementaciones de los lenguajes presentadas en este trabajo.*

Fecha de la presentación: **NOVIEMBRE 2014**

Resumen

Este trabajo es el informe resultado de la tesina de grado de la carrera Licenciatura en Informática. En él se expone mi participación en la creación de PyGobstones 1.0, ambiente de desarrollo implementado en Python que incluye los compiladores y máquinas virtuales de los lenguajes de programación Gobstones 3.0 y XGobstones, lenguajes diseñados para la introducción de las bases conceptuales de la programación.

Agradecimientos

El más grande de los agradecimientos es para mi viejo y mi vieja, que bancaron durante los seis años que involucró el proceso de mi carrera. Sin el amor y el afecto que siempre me dedicaron esto nunca hubiera sido posible.

Agradezco a mis abuelos Julio E. Genovés y Elina C. Pazzelli, monumentos a la sapiencia y al humanismo, quienes fomentaron mi curiosidad desde chico y me llevaron a preguntarme el por qué y el cómo de las cosas.

Agradezco también a mi amigo Facundo Quiroga que fue un gran guía durante mis primeros años en la carrera, siempre dispuesto a contestar mis dudas e inquietudes respecto de distintos ámbitos de la informática, principalmente el desarrollo de videojuegos.

Agradezco a mi gran amigo Rodrigo Oliveri y compañero de la carrera con quién compartí viajes en tren, días completos de estudio, noches completas también (incluso sin dormir), siestas en el bosque y comidas rápidas de lo más insalubres, entre otras cosas.

Un especial agradecimiento a Pablo E. "Fidel" Martínez López quién me deslumbró con sus clases de programación funcional abriéndome las puertas a un mundo completamente nuevo que pareciera pintado por Maurits Cornelis Escher. También le agradezco por haber aceptado participar de mi tesina de licenciatura y haber impulsado mi carrera profesional en el ámbito de la enseñanza de la programación.

Un gran agradecimiento a Gabriel Baum que aceptó dirigir mi tesina de licenciatura.

Agradezco a Pablo Barenbaum que me asistió en mis primeros pasos con PyGobstones, siempre atento a responder mis dudas respecto de la implementación.

Agradezco a toda la gente de la Universidad Nacional de Quilmes, en especial mis colegas de la materia Introducción a la Programación, Eduardo A. Bonelli, Francisco Soulignac, Flavia Saldaña, Pablo Tobia, Valeria De Cristófolo y Pablo Barenbaum, que dieron feedback sobre mis desarrollos y me acompañaron durante el proceso.

Agradezco también a Leonardo Marina por haberme dado ese empujón que me faltaba para terminar la carrera.

También agradezco a mis hermanas, mis primos, mis tíos, mis amigos y a todos los que me acompañaron durante mi carrera.

Por sobre todas las cosas, agradezco la posibilidad de agradecer, que es la manera de apreciar la vida y reconocer las virtudes que nos rodean.

Este documento se creo utilizando el procesador de textos LaTeX.

Índice

1. Introducción

1.1. Contexto

Al considerar la enseñanza de la programación en personas con ningún conocimiento previo, en la Universidad Nacional de Quilmes abordaron el problema a través del diseño de una secuencia didáctica innovadora. Junto a ella también diseñaron el lenguaje de programación GOBSTONES, que captura exactamente la filosofía y las necesidades de dicha secuencia. Este lenguaje fue implementado por distintas herramientas entre las que se destaca PYGOBSTONES 0.97: un ambiente de desarrollo implementado en PYTHON que incluye un compilador de GOBSTONES 2.0 y una interfaz minimalista.

En 2013 se trabajó en una extensión del lenguaje, XGOBSTONES, basado en el original GOBSTONES en base a la necesidad de extender los elementos básicos del lenguaje para incorporar estructuras de datos básicas y otros elementos avanzados (como manejo de referencias). En conjunto con esta nueva extensión se diseñó GOBSTONES 3.0 que aporta nuevas características que mejoran la expresividad del lenguaje y contempla la compatibilidad con la nueva sintaxis de XGOBSTONES. Al momento de iniciar el trabajo aquí reportado no existía una herramienta que implementase estos lenguajes.

Mi trabajo en esta tesina se enmarcó dentro de las actividades del equipo de desarrollo de PYGOBSTONES que fue conformado con el objetivo de desarrollar una nueva versión de esta herramienta, PYGOBSTONES 1.0, que implemente los nuevos lenguajes de programación. Mi participación consistió en ser el responsable de la implementación de los nuevos compiladores, las nuevas máquinas virtuales que ejecutarán el código compilado y un mecanismo de comunicación entre éstos y la nueva interfaz gráfica desarrollada por otros miembros del equipo.

1.2. Problema

La problemática tratada en esta tesis es la falta de una herramienta que implemente los lenguajes de programación GOBSTONES 3.0 y XGOBSTONES.

Como parte del trabajo de implementar esta nueva herramienta se identificaron las siguientes funcionalidades:

- Desarrollo del compilador y máquina virtual de GOBSTONES 3.0.
- Desarrollo del compilador y máquina virtual de XGOBSTONES.
- Desarrollo de la interfaz gráfica.
- Desarrollo de un mecanismo de comunicación entre la interfaz gráfica y los compiladores y la máquinas virtuales que implementan los lenguajes GOBSTONES 3.0 y XGOBSTONES.

Siendo que la interfaz gráfica ya se encontraba en proceso de desarrollo por parte del equipo, esta tesina da respuesta a la problemática relacionada con la

falta de compiladores y máquina virtuales para XGobstones y Gobstones 3.0 y un mecanismo de comunicación entre la interfaz gráfica y éstos.

1.3. Objetivo

Esta tesina centra sus esfuerzos en el desarrollo de compiladores y máquinas virtuales para los lenguajes de programación Gobstones 3.0 y XGobstones y en el diseño e implementación del mecanismo necesario para la interacción entre éstos y la nueva interfaz gráfica de PyGobstones, conformando de esta manera la nueva herramienta PyGobstones 1.0.

En resumen, los objetivos relacionados a esta tesina son:

a) Desarrollo del compilador y la máquina virtual de Gobstones 3.0 a partir del existente compilador y máquina virtual de Gobstones 2.0 incorporando las implementaciones necesarias para soportar las nuevas características del lenguaje.

b) Desarrollo de un compilador y una máquina virtual para el lenguaje de programación XGobstones basado en el compilador y máquina virtual de Gobstones 3.0.

c) Definición de un módulo de interacción para que los compiladores y máquinas virtuales de ambos lenguajes puedan interactuar con la interfaz gráfica de PyGobstones 1.0 preservando el máximo de modularidad.

1.4. Sumario

Comenzaremos introduciendo los antecedentes sobre los cuales se construye esta tesina: el lenguaje de programación Gobstones 2.0 en la sección 2.1 y sus dos implementaciones en las secciones 2.2 y 2.3. Luego continuaremos describiendo los lenguajes implementados, Gobstones 3.0 y XGobstones, en las secciones 3 y 4. Por último describiré las implementaciones realizadas en la sección 5 que incluyen la implementación del lenguaje Gobstones 3.0 (sección 5.2), la implementación del lenguaje XGobstones (sección 5.3), la implementación del modo interactivo (sección 5.4) y la implementación del módulo de interacción (sección 5.5). Finalmente brindaremos conclusiones.

2. Antecedentes

2.1. El lenguaje de programación Gobstones

El lenguaje de programación Gobstones [7] fue ideado por Pablo E. Martínez López y Eduardo A. Bonelli inicialmente para satisfacer las necesidades de los alumnos de la carrera Tecnicatura Universitaria en Programación Informática que se dicta en la Universidad Nacional de Quilmes. Este lenguaje de programación fue pensado en el marco de un primer año de una carrera informática, para aquellas personas que no tienen conocimientos previos de programación,

basándose fundamentalmente en la simplicidad y claridad de conceptos básicos comunes (generalizables) a todos los paradigmas de programación.

Entre los elementos que componen al lenguaje se encuentran un tablero con bolitas, comandos y sus formas de combinación (las estructuras de control) y abstracción (los procedimientos), expresiones y sus formas de combinación (las operaciones elementales) y abstracción (las funciones), tipos, parametrización y otros elementos. El punto de entrada a un programa GOBSTONES 2.0 es un procedimiento llamado `Main` a partir del cual se desarrolla la ejecución del programa, brindando opcionalmente la posibilidad de retornar variables que serán informadas como parte del resultado final.

Los comandos son operaciones que causan efectos en el tablero. La combinación de estos comandos se da a través de elementos que provee el lenguaje como secuenciación, bloques, alternativa condicional (`if-else`), repetición indexada (`repeatWith-in`) y repetición condicional (`while`). A su vez, además de los comandos primitivos provistos por el lenguaje es posible crear nuevos comandos definidos por el usuario a través de la definición de procedimientos.

Las expresiones son los valores definidos por GOBSTONES (números, colores, direcciones, booleanos) y las combinaciones de estos valores utilizando los operadores (`+`, `*`, `-`, `div`, `mod`, etc.) o invocando funciones, que pueden ser provistas por el lenguaje (e.g. `siguiente(x), previo(x)`) o definidas por el usuario.

El tablero, las bolitas y el cabezal son utilizados por las didácticas que emplean GOBSTONES para dar los primeros pasos en el aprendizaje de los mecanismos de abstracción a través de elementos concretos. El tablero es una cuadrícula de celdas sobre las que opera un cabezal que se desplaza colocando o sacando bolitas de colores en cada celda (ver figura 2.1.1). El cabezal entiende cinco comandos básicos que causan efectos sobre el tablero: `Poner`, que dado un color `c` pone una bolita de dicho color en la celda actual, `Sacar`, que dado un color `c` saca una bolita de dicho color en la celda actual, `Mover`, que dado una dirección `d` mueve el cabezal a la celda lindante en esa dirección, `IrAlBorde`, que dado una dirección `d` mueve al cabezal al borde `d` del tablero, y `VaciarTablero`, que vacía el tablero. A su vez, el cabezal provee de expresiones para conocer el estado del tablero, tales como: `puedeMover`, que dado una dirección `d` indica si el cabezal puede moverse en esa dirección sin caerse del tablero, `nroBolitas`, que dado un color `c` denota la cantidad de bolitas de dicho color en la celda actual, y `hayBolitas`, que dado un color `c` indica si hay alguna bolita de dicho color en la celda actual. GOBSTONES distingue cuatro colores de bolitas diferentes: `Azul`, `Negro`, `Rojo` y `Verde`, y las cuatro direcciones cardinales básicas: `Norte`, `Este`, `Sur`, `Oeste`.

La versión del lenguaje utilizada antes de la realización de esta tesina es GOBSTONES 2.0. Por esta razón presento los ejemplos de esta sección en dicha versión, para luego contar en la sección siguiente los cambios introducidos por GOBSTONES 3.0.

Como ejemplo de un primer programa en GOBSTONES 2.0 mostramos uno que pone una bolita de cada color y mueve al norte.

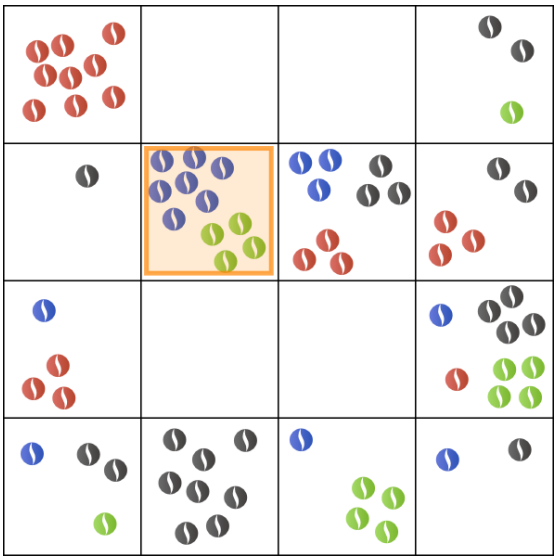


Figura 2.1.1: Representación simple del tablero. Las bolitas se representan como circulos coloreados. El cabezal se muestra en color naranja.

```
procedure Main()
 {
   Poner(Verde)
   Poner(Rojo)
   Poner(Negro)
   Poner(Azul)
   Mover(Norte)
 }
```

Un ejemplo de utilización de procedimientos es `PonerN`, que dados un color `c` y un número `n` coloca `n` bolitas de color `c` en la celda actual. Se muestra la utilización de este procedimiento en el siguiente programa que coloca veinte bolitas de color rojo.

```
procedure PonerN(c, n)
 {
   repeatWith i in 1..n
    { Poner(c) }
 }

procedure Main()
 { PonerN(Rojo, 20) }
```

La implementación de `PonerN` se logra poniendo bolitas de color `c` utilizando la repetición indexada (`repeatWith-in`) para iterar sobre un rango `1..n`. El procedimiento `Main` hace uso de `PonerN` para poner veinte bolitas de color rojo

en la celda actual.

Un ejemplo de definición de una función es `nroTotalDeBolitas` que denota la cantidad total de bolitas que hay en la celda actual. El siguiente programa, que pone tantas verdes como bolitas haya en la celda, ilustra la utilización de esta función (además del uso de variables):

```
function nroTotalDeBolitas()
 {
   contador := 0
   repeatWith c in minColor()..maxColor()
    { contador := contador + nroBolitas(c) }
   return(contador)
 }


procedure Main()
 { PonerN(Verde, nroTotalDeBolitas()) }
```

Es posible observar la definición de la función `nroTotalDeBolitas` que denota la cantidad de bolitas de cualquier color presentes en la celda actual. Esta misma se implementa con un contador y la repetición indexada sobre el rango `minColor()..maxColor()` para recorrer todos los colores definidos por GOBSTONES 2.0 donde para cada color se incrementa el contador con la cantidad de bolitas de dicho color presentes en la celda. Esta función se utiliza en el contexto del procedimiento `Main` como argumento de `PonerN` generando el efecto descripto anteriormente.

El procedimiento `Main` es capaz de retornar variables de manera opcional. El siguiente programa asigna cuatro valores con expresiones de distintos tipos y las retorna.

```
procedure Main()
 {
   n := 3 * 10
   c := Verde
   d := Norte
   b := True
   return(n, c, d, b)
 }
```

El resultado de la ejecución del ejemplo anterior es el siguiente:

```
n -> 30
c -> Verde
d -> Norte
b -> True
```

En el apéndice H.1 se presenta un ejemplo de un programa complejo en GOBSTONES 2.0.

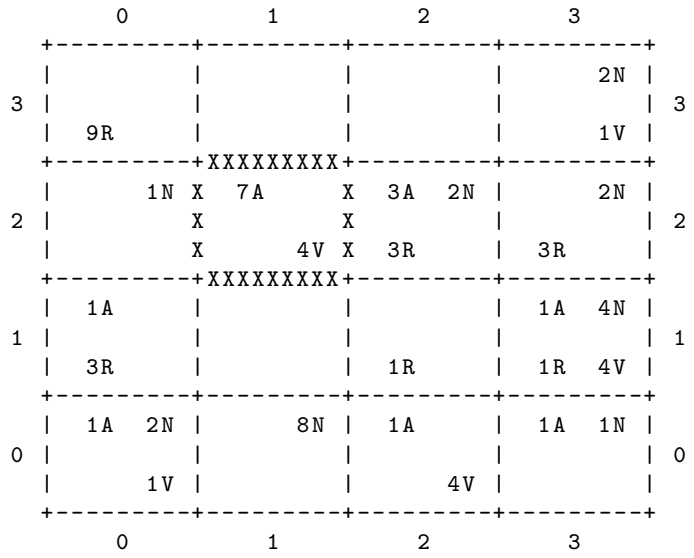


Figura 2.2.1: Tablero en formato ASCII.

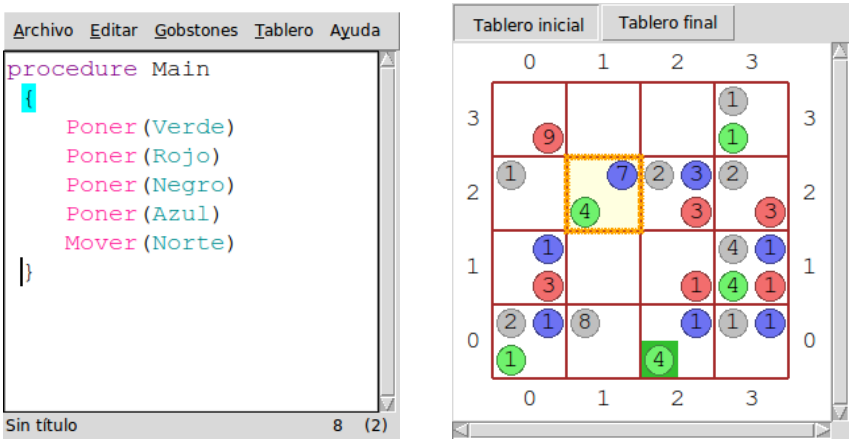
2.2. Implementación en Haskell de Gobstones 2.0

La primer implementación de Gobstones fue desarrollada por Pablo E. Martínez López en el lenguaje de programación funcional Haskell como un intérprete. La interacción con el usuario se realizaba a través de la línea de comandos provista por el intérprete de Hugs y la edición de tableros se realizaba a mano sobre un archivo de texto plano. En la figura 2.2.1 se puede observar el tablero representado en la figura 2.1.1 tal como se habría visualizado con esta herramienta (de haber tenido visualización para tableros 4×4).

Debido a los importantes problemas de eficiencia y usabilidad ligados al intérprete de Haskell (por ejemplo, el tablero solo podía ser de 8×8 celdas debido al método de dibujo utilizado) rápidamente se lo reemplazó por PyGobstones 0.97.

2.3. La herramienta PyGobstones 0.97

La herramienta PyGobstones 0.97 es un ambiente de desarrollo, implementado en Python por Pablo Barenbaum, que incluye un compilador de la versión 2.0 de Gobstones, una máquina virtual que ejecuta el código compilado y una interfaz gráfica minimalista. Esta herramienta constituyó una importante mejora con respecto a la implementación desarrollada en Haskell en dos aspectos: por un lado, mejora significativamente la usabilidad al proporcionar una interfaz gráfica adecuada (ver figura 2.3.1) y, por otro lado, presenta una importante mejora en el rendimiento en la ejecución de programas.



(a) Editor de programas. **(b)** Editor de tablero.

Figura 2.3.1: Interfaz gráfica de PYGOBSTONES 0.97.

Las mejoras en la interfaz gráfica incluyen un espacio de edición de código con *syntax highlighting* de acuerdo a la especificación GOBSTONES 2.0, un editor de tableros que permite, entre otras cosas, la generación automática de tableros aleatorios y otros elementos experimentales.

La verdadera mejora para la eficiencia que incorpora PYGOBSTONES 0.97 es el cambio de la interpretación de GOBSTONES 2.0 a un proceso de compilación [1]. Es relevante a los objetivos de esta tesina analizar la arquitectura de módulos de la implementación del compilador del lenguaje GOBSTONES 2.0 que forma parte de esta herramienta, puesto que mi trabajo se construye sobre él.

2.3.1. Estructura del compilador y máquina virtual

El compilador provisto por PYGOBSTONES 0.97 se estructura en tres etapas que debe atravesar un programa de GOBSTONES 2.0: análisis sintáctico, análisis semántico y compilación. A partir del *bytecode*[1]generado en la etapa de compilación se procede a su ejecución en la máquina virtual. La figura 2.3.2 ilustra el proceso completo.

En la etapa de análisis sintáctico se realiza dicho análisis sobre el programa provisto como entrada. Para ello, el analizador sintáctico carga adicionalmente una gramática BNF [5], codificada en un archivo de texto plano, que define la gramática de GOBSTONES 2.0 (descripta en el apéndice F). A partir de esta gramática y del programa, este módulo genera un árbol de sintaxis abstracta.

[1]El término bytecode refiere a una forma de código intermedia entre un lenguaje de alto nivel y el assembler de una arquitectura específica, y que usualmente se ejecuta sobre una máquina virtual para proveer portabilidad. El nombre hace referencia a que originalmente cada código de operación se codificaba en un byte, y aunque actualmente no sea así, el nombre se popularizó para este tipo de código intermedio.

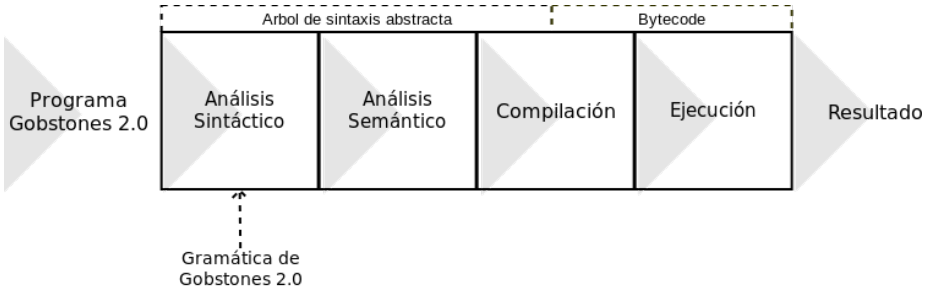


Figura 2.3.2: Pipeline del compilador y máquina virtual incorporados en PyGobstones 0.97

En la etapa de análisis semántico se verifican condiciones estructurales vinculadas al árbol de sintaxis abstracta. Dependiendo de los parámetros utilizados al momento de la invocación del compilador y máquina virtual, esta etapa puede realizar el chequeo de tipos y de alcance de los identificadores del programa.

Durante la etapa de compilación se compila el árbol de sintaxis abstracta en el bytecode definido para PyGobstones 0.97, de manera que las definiciones, comandos y expresiones del lenguaje sean representados en base a un conjunto de primitivas que faciliten su ejecución.

Finalmente, en la etapa de ejecución, el bytecode generado es ejecutado por la máquina virtual (se describe en la sección 2.3.3).

2.3.2. El bytecode de Gobstones

El bytecode de Gobstones se compone de un conjunto de instrucciones primitivas adecuadas para su ejecución en la máquina virtual. Gran parte de estas instrucciones hacen uso de la pila del programa y poseen parámetros asociados: `pushVar` y `pushConst` apilan valores en la pila del programa, `jump`, `jumpIfFalse` y `jumpIfNotIn` proveen saltos a etiquetas presentes en el programa (dispuestas mediante una instrucción `label`), `call` provee un salto a una rutina con cambio de contexto, `procedure`, `function`, `end`, `enter`, `leave`, `return` y `returnVars` constituyen instrucciones para la indicación de entrada y salida a rutinas y `assign` que asocia un valor a un nombre.

El siguiente es un ejemplo de compilación de un programa que define el procedimiento `IrAlExtremo`, que dado una dirección `dir` lleva al cabezal al extremo `dir` del tablero, que es invocado en el procedimiento `Main` con la dirección `Sur` como argumento.

```
procedure IrAlExtremo(dir)
 {
   while (puedeMover(dir))
    { Mover(dir) }
 }
```

```
procedure $IrAlExtremo dir
  label L_6309194320
  pushVar dir
  call puedeMover 1
  jumpIfFalse L_6309196496
  pushVar dir
  call Mover 1
  jump L_6309194320
  label L_6309196496
  return 0
end
```

```
procedure Main()
 { IrAlExtremo(Sur) }
```

```
procedure $Main
  pushConst Sur
  call $IrAlExtremo 1
  returnVars 0
end
```

Podemos observar que la estructura de repetición condicional del procedimiento `IrAlExtremo` se compila en una combinación de instrucciones de salto responsables de emular su comportamiento. Podemos ver también la presencia de instrucciones `call` que indican a la máquina virtual invocaciones de procedimientos y funciones.

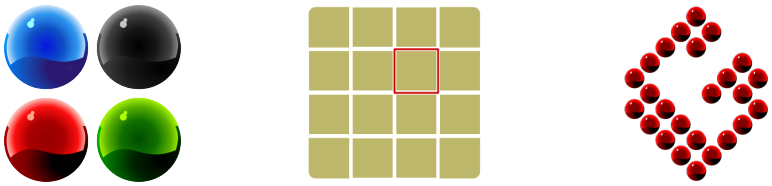
En el apéndice D se realiza una descripción exhaustiva del bytecode de GOBSTONES.

2.3.3. La máquina virtual de PYGOBSTONES 0.97

La máquina virtual de PYGOBSTONES 0.97 posee las implementaciones para cada una de las instrucciones del bytecode y las implementaciones de las primitivas (funciones y procedimientos) que se encuentran disponibles para el programador sin definición previa (e.g. `Poner`, `nroBolitas`, `siguiente`, etc.). También es la responsable de la representación de todos los valores definidos por GOBSTONES 2.0 y de la representación del tablero.

Esta máquina virtual funciona manteniendo una pila de ejecución donde se almacenan los registros de activación de cada rutina, compuestos por un puntero de instrucción (*ip*), una pila de programa y variables, y un estado global, que almacena el estado del tablero. La pila de programa es utilizada durante la invocación de funciones y procedimientos para almacenar los valores que serán argumento de la llamada y, luego de la llamada, para almacenar el resultado de las funciones.

La máquina virtual permite la implementación de operaciones internas anteponiendo un guión bajo al nombre de la operación (e.g. `_read`), siendo inaccesibles para el programador. Si bien esta característica no es utilizada en PYGOBSTONES 0.97, se hace uso de ella para las nuevas implementaciones que desarrolla esta tesina.



(a) Bolitas de colores **(b)** Tablero y cabezal **(c)** Logotipo

Figura 2.4.1: Identidad visual de PyGobstones 1.0.

Figura 2.4.2: Representación de un tablero con bolitas según la nueva representación utilizada por PyGobstones 1.0.

2.4. La herramienta PyGobstones 1.0

Al momento de comenzar esta tesina había varios aspectos de la herramienta PyGobstones 1.0 que ya se encontraban resueltos: la identidad visual y el módulo que implementa la interfaz gráfica.

Un avance importante en la herramienta PyGobstones 1.0 fue la definición de una identidad visual para la interfaz gráfica. Esto incluyó la reformulación de la representación del tablero. En la representación elegida las bolitas se visualizan como esferas de colores (ver figura 2.4.1a) mientras que el tablero es una grilla con bordes redondeados donde cada celda es de color amarillo oscuro y donde el cabezal se muestra como un línea de color rojo que contornea la celda actual (ver figura 2.4.1b). Estos elementos se combinan en un tablero con bolitas, donde en cada celda se visualiza a lo sumo una bolita por cada color y donde la cantidad de bolitas de un color dado, presentes en una celda particular, se indica con un número que se encuentra centrado en la imagen de la bolita. En la figura 2.4.2 se muestra la representación para un tablero con bolitas.

El logotipo mostrado en PyGobstones 1.0, es el elegido para Gobstones 3.0, consistente en una letra “G” construida con bolitas rojas como se puede

ver en la figura 2.4.1c. Este logo cambia a bolitas de color azul para identificar al lenguaje XGOBSTONES.

La interfaz gráfica se programó en PYTHON utilizando la biblioteca gráfica QT y consta de una ventana principal, que contiene al editor de programas y un menú con botones para acceder a las distintas funcionalidades provistas, y dos ventanas adicionales para la edición de tableros y la visualización de los resultados de una ejecución.

El editor de programas (figura 2.4.3a) es un editor de texto extendido que provee ayudas visuales para el programador tales como *syntax highlighting* para los elementos de los lenguajes de programación GOBSTONES 3.0 y XGOBSTONES y los números de línea.

El editor de tableros (figura 2.4.3b) es una herramienta que permite la edición de tableros. Este editor permite agregar y sacar bolitas al tablero, cambiar la posición del cabezal, cambiar el tamaño del tablero, vaciar el tablero, generar bolitas de manera aleatoria sobre éste y establecer un tablero como tablero inicial a ser utilizado en las ejecuciones. Además permite guardar un tablero en un archivo de texto plano en disco, según el formato GBB v1.0 (ver apéndice I), y cargar un tablero desde estos archivos.

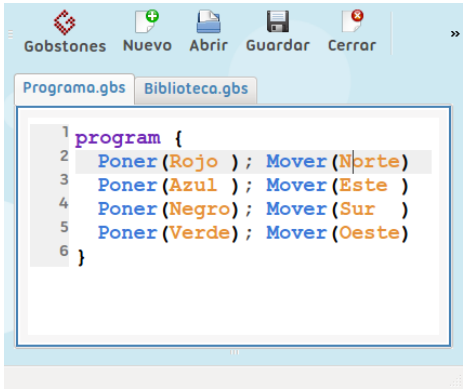
La vista de resultados (2.4.3c) muestra el resultado de una ejecución particular. Esta vista presenta inicialmente el tablero final resultado de la ejecución permitiendo alternar a otras pestañas que contienen el tablero inicial empleado en la ejecución, el programa GOBSTONES 3.0 o XGOBSTONES utilizado para la transformación del tablero y aquellas expresiones que fueron retornadas al finalizar la ejecución.

Parte de los esfuerzos de esta tesina se centra en la creación de un módulo de interacción que permita integrar esta interfaz gráfica con las implementaciones de los lenguajes GOBSTONES 3.0 y XGOBSTONES que desarrollé en el transcurso de la misma.

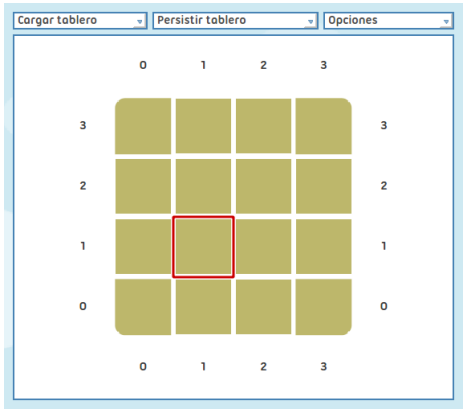
3. El lenguaje de programación GOBSTONES 3.0

El lenguaje de programación GOBSTONES 3.0 es una modificación de la versión 2.0 de GOBSTONES que incorpora diversos cambios sintácticos (incluidos en el apéndice B) y semánticos, de manera que el lenguaje se adapte para ofrecer mejoras en la secuencia didáctica y, adicionalmente, para ser extendido, por ejemplo, a XGOBSTONES. Entre los cambios más destacados podemos encontrar una mejora en la forma de definir un programa completo conceptualmente menos compleja que la de GOBSTONES 2.0, una nueva forma de repetición conceptualmente más simple (la repetición simple), un comportamiento extendido de la repetición indexada (que permite especificar secuencias explícitas y el incremento de los rangos) y el modo interactivo, que constituye un mecanismo que permite al usuario interactuar con el programa obteniendo resultados parciales de la ejecución.

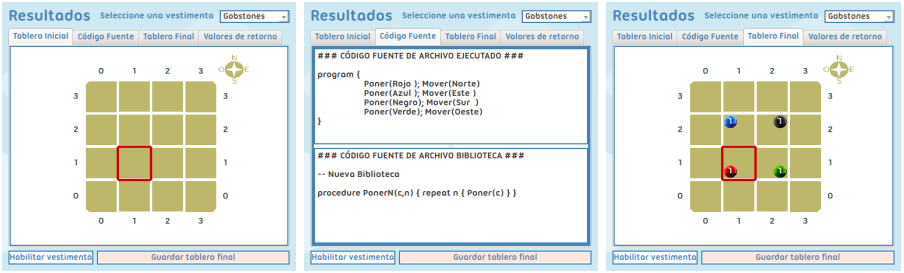
Realicé diversas colaboraciones al diseño de GOBSTONES 3.0, principalmente en la definición del modo interactivo.



(a) Editor de programas.



(b) Editor de tableros.



(c) Vista de resultados.

Figura 2.4.3: Interfaz de PYGOBSTONES 1.0

3.1. Cambios necesarios en la secuencia didáctica

El lenguaje de programación GOBSTONES 3.0 introduce varios cambios de sintaxis respecto de su predecesor, GOBSTONES 2.0. En esta sección enumeramos y ejemplificamos cada uno de estos cambios.

Una de las incorporaciones más importantes de GOBSTONES 3.0 a nivel conceptual es la incorporación de la nueva definición `program` como punto de entrada del programa en reemplazo del anterior `procedure Main`. Este cambio, si bien simple, favoreció la secuencia didáctica permitiendo posponer la presentación de la noción de procedimientos y reduciendo la cantidad de conceptos involucrados en los primeros contactos con el lenguaje.

```
// Gobstones 2.0
procedure Main()
 { /* instrucciones */ }
```

```
// Gobstones 3.0
program
 { /* instrucciones */ }
```

Por otro lado, GOBSTONES 3.0 establece un nueva forma de repetición que evita la utilización de un índice: la repetición simple, `repeat`. Esto implica una mejora en la secuencia didáctica dado que es posible enseñar el concepto de la repetición de comandos demorando la explicación de identificador, índice y rangos para estadios posteriores del aprendizaje. A continuación se muestran dos procedimientos en su versión GOBSTONES 2.0 y en su versión GOBSTONES 3.0, utilizando la repetición simple.

```
// Gobstones 2.0
procedure Mover10AlNorte()
 {
   repeatWith i in 1..10
    { Mover(Norte) }
 }
```

```
// Gobstones 3.0
procedure Mover10AlNorte()
 {
   repeat (10)
    { Mover(Norte) }
 }
```

```
// Gobstones 2.0
procedure PonerN(c, n)
 {
   repeatWith i in 1..n
    { Poner(c) }
 }
```

```
// Gobstones 3.0
procedure PonerN(c, n)
 {
   repeat (n)
    { Poner(c) }
 }
```

La repetición indexada fue objeto de un par de cambios. El primero de ellos es que su nombre, anteriormente `repeatWith-in`, es cambiado a `foreach-in`, incorporando además corchetes para delimitar la secuencia sobre la cual se itera. Este cambio obedece a una estandarización de las palabras reservadas, tendientes a mejorar la transición a otros lenguajes en cursos posteriores, y a un uso ortogonal de las secuencias (ver el cambio que se describe a continuación).

```
// Gobstones 2.0
procedure Poner10Verde()
 {
   repeatWith i in 1..10
    { Poner(Verde) }
 }
```

```
// Gobstones 3.0
procedure Poner10Verde()
 {
   foreach i in [1..10]
    { Poner(Verde) }
 }
```

```
// Gobstones 2.0
procedure PonerN(c, n)
 {
   repeatWith i in 1..n
    { Poner(c) }
 }
```

```
// Gobstones 3.0
procedure PonerN(c, n)
 {
   foreach i in [1..n]
    { Poner(c) }
 }
```

El segundo cambio para esta estructura es permitir especificar el incremento de los rangos y permitir la definción de secuencias explícitas. Mostramos diversos ejemplos esquematizando los nuevos comportamientos.

```
procedure Trazar(i)
 /*
   PROPóSITO
     Dibuja un segmento de una linea
     poniendo i bolitas de color Verde.
   PRECONDICIóN
     Hay al menos i celdas en dirección
     Este y/o Norte.
   OBSERVACIóN
     Es un procedimiento auxiliar para los
     ejemplos que siguen.
 */
 {
   PonerN(Verde, i)
   if (puedeMover(Este))
    { Mover(Este) }
   else
    { Mover(Norte)
      IrAlBorde(Oeste) }
 }
```

```
program
 /*
   PRECONDICIóN
     El tablero mide 10x7. El cabezal se encuentra
     en la posición (0,0).
```

```
    OBSERVACIóN
      El resultado de cada foreach se visualizará
      en una fila, gracias a la precondición.
  */
  {
    VaciarTablero()

    # Rango simple
    foreach i in [1..10]
     { Trazar(i) }

    # Rango con incremento 2
    foreach i in [2,4..20]
     { Trazar(i) }

    # Rango con decremento 1
    foreach i in [30,27..1]
     { Trazar(i) }

    # Secuencia explícita de números
    foreach i in [14,11,2,3,4,13,1,20,3,7]
     { Trazar(i) }

    # Secuencia explícita de direcciones
    foreach d in [Norte, Norte, Este, Este, Sur, Este]
     { Poner(Rojo); Mover(d) }
  }
```

La figura 3.1.1 muestra cada una de las trazas creadas por el ejemplo donde el número de bolitas denota el valor del índice de la repetición indexada en una iteración particular, siendo la celda del borde oeste el valor del índice para la primer iteración y la celda del borde este el valor del índice para la última iteración, de abajo para arriba: la primer fila muestra el resultado de la traza del ejemplo de rango simple, la segunda fila el ejemplo de rango con incremento dos, la tercera el ejemplo de rango con decremento una, la cuarta el ejemplo de secuencia explícita de números, las bolitas rojas muestran el recorrido generado con el ejemplo de la secuencia explícita de direcciones que comienza en el borde oeste de la quinta fila.

3.2. Otros cambios para compatibilidad

Además de los cambios mencionados en la sección anterior, el lenguaje incorpora otros cambios menores con el fin de lograr compatibilidad con la sintaxis del lenguaje de programación XGOBSTONES.

La estructura `case-of` de alternativa indexada de GOBSTONES 2.0 se renombró a `switch-to` en GOBSTONES 3.0. Este cambio libera la palabra reservada `case` para su uso diferente en XGOBSTONES. El siguiente es un ejemplo

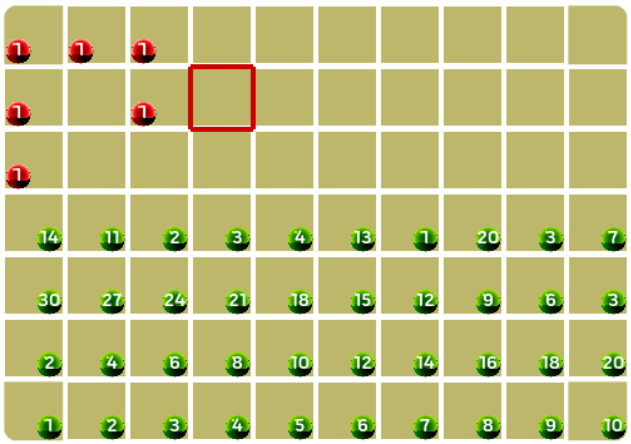


Figura 3.1.1: Representación del tablero utilizada en PYGOBSTONES 1.0. Tablero resultante del programa que ilustra el comportamiento de la repetición indexada en GOBSTONES 2.0.

escrito en ambas versiones del lenguaje, donde se muestra un procedimiento que pone una bolita dependiendo del número denotado por el parámetro `n`.

```
// Gobstones 2.0
procedure PonerNumero(n)
 {
   case (n) of
    {
      1 -> { Poner(Azul) }
      2 -> { Poner(Negro)}
      3 -> { Poner(Rojo) }
      _ -> { Poner(Verde)}
    }
 }
```

```
// Gobstones 3.0
procedure PonerNumero(n)
 {
   switch (n) to
    {
      1 -> { Poner(Azul) }
      2 -> { Poner(Negro)}
      3 -> { Poner(Rojo) }
      _ -> { Poner(Verde)}
    }
 }
```

También se realizaron modificaciones en la estructura `if-else` a la que se le agregó la palabra clave (opcional) `then`. Mostramos dos ejemplos que reflejan el cambio en la estructura con respecto a la versión anterior del lenguaje.

```
// Gobstones 2.0
procedure SacarSiHay(c)
 {
   if (hayBolitas(c))
    { Sacar(c) }
 }
```

```
// Gobstones 3.0
procedure SacarSiHay(c)
 {
   if (hayBolitas(c))
    then { Sacar(c) }
 }
```

```
// Gobstones 2.0
procedure InvertirColor()
 {
   if (hayBolitas(Rojo))
    { Sacar(Rojo)
      Poner(Azul) }
    else
     { Sacar(Azul)
       Poner(Rojo) }
 }
```

```
// Gobstones 3.0
procedure InvertirColor()
 {
   if (hayBolitas(Rojo))
    then
     { Sacar(Rojo)
       Poner(Azul) }
    else
     { Sacar(Azul)
       Poner(Rojo) }
 }
```

Otro cambio de sintaxis fue el agregado de comentarios al estilo PYTHON (además de los existentes estilo C y HASKELL) definidos por el carácter `#` para indicar el comienzo de un comentario de línea y el delimitador `"""` para delimitar los comentarios multi-línea, como se muestra en el siguiente ejemplo:

```
# Este es un comentario de una sola línea.

""" Este es
  un comentario
  multi-línea """
```

3.3. Modo interactivo

Esta nueva versión de GOBSTONES incorpora un modo interactivo al lenguaje que permite al usuario interactuar con el programa en un ciclo *read-eval-print*, donde luego de cada tecla presionada se ejecuta algún código y se muestra el tablero así generado, quedando a la espera de nuevas teclas. Esto posibilita, entre otras cosas, el desarrollo de pequeños videojuegos por turnos.

La utilización de este modo se indica a través del uso de las palabras clave `interactive program`, que indica que el programa se compondrá de la especificación de una serie de alternativas $(teclas, bloque)$ para asociar un conjunto de teclas a un bloque específico de código (incluyendo una alternativa por defecto). Estas asociaciones serán utilizadas al momento de consumir las entradas de teclado ingresadas por el programador durante la fase *read* del ciclo *read-eval-print* para decidir qué debe ser ejecutado en cada iteración (en la fase *eval* del ciclo).

En el siguiente ejemplo se muestra la utilización del modo interactivo en un programa que permite la manipulación del cabezal del tablero durante la ejecución. Este programa pone bolitas utilizando la tecla que corresponde a la inicial del color, las saca utilizando la tecla correspondiente a la inicial de color en combinación con la tecla `SHIFT` y desplaza el cabezal mediante las flechas del teclado.

```
interactive program
 {
   K_A           -> { Poner(Azul)  }
   K_SHIFT_A     -> { Sacar(Azul)  }
   K_N           -> { Poner(Negro) }
   K_SHIFT_N     -> { Sacar(Negro) }
   K_R           -> { Poner(Rojo)  }
   K_SHIFT_R     -> { Sacar(Rojo)  }
   K_V           -> { Poner(Verde) }
   K_SHIFT_V     -> { Sacar(Verde) }
   K_ARROW_UP    -> { Mover(Norte) }
   K_ARROW_DOWN  -> { Mover(Sur)   }
   K_ARROW_RIGHT -> { Mover(Este)  }
   K_ARROW_LEFT  -> { Mover(Oeste) }
   _             -> { Skip }
 }
```

Observar la forma en que se especifican las teclas mediante constantes predefinidas prefijadas con `K_`, y como cada tecla se asocia con un bloque de código.

En el apéndice H.2 se presenta un pequeño juego programado en GOBSTONES 3.0 utilizando el modo interactivo.

4. El lenguaje de programación XGOBSTONES

El lenguaje de programación XGOBSTONES es una extensión de el lenguaje GOBSTONES 3.0 que incorpora estructuras de datos básicas y opcionalmente las nociones de memoria y pasaje de parámetros por referencia. La sintaxis de este lenguaje se encuentra en el apéndice C.

Participé durante el diseño de XGOBSTONES aportando ideas respecto de las interfaces de las estructuras de datos y las capacidades de los tableros en su nuevo rol como elemento explícito.

4.1. Estructuras de datos básicas

Esta extensión incorpora a GOBSTONES 3.0 las estructuras de datos lista, registro y variante, con sus respectivos mecanismos de generación y de acceso.

Es interesante destacar que, en coherencia con la filosofía de GOBSTONES de denotar valores mediante expresiones, las estructuras de datos son valores puros que se describen a través de las operaciones que se presentan en esta sección. En otras palabras, no hace falta incorporar la noción de memoria o de representación para presentar este concepto.

En esta sección hago una descripción exhaustiva de cada una las estructuras de datos que forman parte de XGOBSTONES.

4.1.1. Listas

Entre las estructuras de datos que agrega XGOBSTONES podemos encontrar las listas, que son simplemente secuencias de elementos del lenguaje. El lenguaje posee expresiones para su generación, acceso y para determinar su nulidad. Para la generación de listas se incorporan tres expresiones básicas: la lista vacía, la lista con un único elemento y la concatenación.

```
program
 {
   xs := []          # Lista vacía
   ys := [1]         # Lista con un único elemento
   zs := ys ++ xs    # Concatenación de dos listas
 }
```

Adicionalmente, el lenguaje provee dos generadores de listas avanzados: el generador de secuencias explícitas y el generador de rango. El generador de secuencias explícitas es una expresión que permite crear una lista a partir de la enumeración de sus elementos. El generador de rango crea una lista a partir de un número inicial y un número final, especificando opcionalmente el incremento o decremento. Estas construcciones siguen la misma sintaxis que las secuencias de la construcción `foreach-in` de GOBSTONES 3.0, uniformizando además dicha forma de repetición para permitir en XGOBSTONES cualquier expresión que denote listas.

```
program
 {
   /* Secuencias explícitas no-numéricas */
   cs := [Verde, Verde, Rojo, Negro]
   ds := [Norte, Norte, Este, Sur, Este]
   bs := [True, True, True, False, False]

   /* Secuencias explícitas numéricas */
   xs  := [1,4,7,10,13,16]
   ys  := [1,2,3,4,5,6,7,8,9,10]
   zs  := [14,12,10,8,6,4,2]

   /* Rangos */
   xsr := [1,4..16]      # [1,4,7,10,13,16]
   ysr := [1..10]        # [1,2,3,4,5,6,7,8,9,10]
   zsr := [14,12..2]     # [14,12,10,8,6,4,2]

   /* Repetición indexada usando expresiones de
      listas no constantes. */
   foreach c in cs
    { Poner(c) }
 }
```

Observar la generación de listas de colores (`cs`), de direcciones (`ds`), de booleanos

(`bs`) y numéricas (`xs`, `ys`, `zs`) a partir de secuencias explícitas, la generación de listas a partir de un rango con incremento (`xsr`), un rango sin incremento (`ysr`) y un rango con decremento (`zsr`), y la utilización de la repetición indexada utilizando expresiones de listas no constantes, que en este ejemplo es `cs`, lo que a partir de un elemento `c` de esta lista, que denota un color en cada iteración, tiene el efecto de poner una bolita de dicho color `c` en la celda actual.

Sobre una lista se pueden aplicar expresiones que permiten denotar elementos componentes de la misma: `head`, que denota el primer elemento de la lista, `tail`, que denota la cola de la lista, `last`, que denota el último elemento de la lista, e `init`, que denota el comienzo de la lista (todos los elementos con excepción del último).

```
program
 {
   xs       := [1,2,3,4]
   cabeza   := head(xs)     # 1
   ultimo   := last(xs)     # 4
   cola     := tail(xs)     # [2,3,4]
   comienzo := init(xs)     # [1,2,3]
 }
```

Además de las expresiones generadoras y de acceso, es posible determinar si una lista no contiene elementos utilizando la expresión `isEmpty`.

```
program
 {
   xs := []
   ys := [1,2,3,4]
   emptyXs := isEmpty(xs)     # True
   emptyYs := isEmpty(ys)     # False
 }
```

Mediante los elementos presentados es posible recorrer una lista o bien mediante la repetición condicional, utilizando las expresiones `head`, `tail` y `isEmpty`, o bien mediante la repetición indexada sobre listas, según la extensión explicada.

```
function sumarListaR(ns)
 {
   rec  := ns
   suma := 0
   while (not isEmpty(rec))
    {
      suma := suma + head(rec)
      rec  := tail(rec)
    }
   return (suma)
 }
```

```
function sumarListaF(ns)
 {
   suma := 0
   foreach n in ns
    { suma := suma + n }
   return (suma)
 }
```

4.1.2. Registros

Otra estructura de datos que agrega XGOBSTONES son los registros, una agrupación de valores nombrados (mediante campos). El lenguaje provee sintaxis para la definición de diferentes tipos de registros por parte del usuario y adicionalmente expresiones para su generación, acceso y modificación.

Un tipo de registro se define utilizando las palabras claves `type-is-record` y especificando cada uno de los campos que compondrán al registro. Ejemplificamos con un tipo particular de registros: una versión simple para la representación de personas.

```
type Persona is record
 {
   field nombre
   field dni
   field edad
 }
```

Esta declaración establece un nuevo tipo llamado `Persona` compuesto por los campos `nombre`, `dni` y `edad`. Observar que los campos no presentan un tipo asociado y que se indican explícitamente mediante la palabra clave `field` acorde a la filosofía de GOBSTONES de identificar cada concepto.

Dada la definción de un registro, existe una expresión para la generación de un dato con ese tipo: el constructor, que se invoca utilizando el nombre con el cual se definió al registro. Mediante esta expresión es posible generar un nuevo registro con los valores deseados para cada campo.

```
program
 {
   p := Persona(nombre <- "Juan",
                dni    <- "14033002",
                edad   <- 32)
   q := Persona(nombre <- "Maria",
                dni    <- "22345123"
                edad   <- 28)
 }
```

En el ejemplo anterior se puede apreciar la construcción de dos registros de tipo `Persona` a través del constructor del tipo y una sucesión de asociaciones campo-valor, cada una de ellas indicada mediante el operador binario de asociación de campos (`<-`): la variable `p` se asigna con un registro `Persona` donde los campos `nombre`, `dni` y `edad` denotan los valores `"Juan"`, `"14033002"` y `32` respectivamente y la variable `q` se asigna con otro registro `Persona` donde el campo `nombre` denota `"Maria"`, el campo `dni` denota `"22345123"` y el campo `edad` denota `28`.

A partir de la definición del registro, XGOBSTONES genera automáticamente funciones observadoras de los campos nombradas a partir de éstos.

```
program
 {
   p := Persona(nombre <- "Juan",
                dni    <- "14033002",
                edad   <- 32)
   q := Persona(nombre <- "Maria",
                dni    <- "22345123"
                edad   <- 28)

   dniP    := dni(p)        # "14033002"
   nombreQ := nombre(q)     # "Maria"
   edadP   := edad(p)       # 32
 }
```

Observar la utilización sin previa definición explícita de las funciones `dni`, `nombre` y `edad` que denotan los valores de los campos `dni`, `nombre` y `edad` de un registro `Persona` dado.

En la versión básica los registros no son modificables pues son valores. En caso de querer un registro similar a otro, pero donde solo varía un subconjunto de campos, existe una forma sintáctica que permite reutilizar el constructor de registros para tal fin. Esta construcción se logra utilizando el mismo constructor de registro pero pasándole como argumento un registro existente. El siguiente ejemplo, similar al anterior, muestra la utilización de esta característica.

```
program
 {
   r1 := Persona(nombre <- "Javier",
                 dni    <- "0",
                 edad   <- 20)
   nombreR1 := nombre(r)                        # "Javier"

   r2 := Persona(r1 | nombre <- "Ramiro")
   nombreR2 := nombre(r1)                       # "Ramiro"

   r3 := Persona(r2 | dni <- "4433353")
   dniR3 := dni(r3)                             # "4433353"

   r4 := Persona(r3 | edad <- 44)
   edadR4 := edad(r4)                           # 44
 }
```

A partir de un registro `Persona r1`, con nombre `"Javier"`, DNI igual a cero y edad igual a 20, actualizo primero el nombre, luego el DNI y por último la edad.

El siguiente ejemplo muestra la definición de la función `crecer` que recibe una persona y retorna una nueva persona con la edad incrementada en uno.

```
function crecer(p)
 { return(Persona(p | edad <- edad(p) + 1)) }

program
 {
    r  := Persona(nombre <- "Javier",
                  dni    <- "35456489",
                  edad   <- 20)
    r' := crecer(r)

    edadR  := edad(r)      # 20
    edadR' := edad(r')     # 21
 }
```

La función `crecer` se define a partir del constructor `Persona` y una persona `p` existente. Esta función se utiliza en el programa principal para aumentar la edad de `r`, asignando el nuevo valor a un registro `r'`. Observar que la función `crecer` retorna un valor distinto al de su argumento.

Es posible crear registros de mayor complejidad que contengan estructuras de datos. Mostramos un ejemplo de un registro `Curso` que relaciona a un docente (registro de tipo `Persona`) con un conjunto de alumnos (lista de registros de tipo `Persona`).

```
type Curso is record
 {
   field docente
   field alumnos
 }

program
 {
   pedro  := Persona(nombre <- "Pedro",
                     dni    <- "30303203",
                     edad   <- 20)
   maria  := Persona(nombre <- "Maria",
                     dni    <- "30305503",
                     edad   <- 18)
   javier := Persona(nombre <- "Javier",
                     dni    <- "20303203",
                     edad   <- 35)

   curso := Curso(docente  <- javier,
                  alumnos  <- [maria, pedro])
   return(curso)
 }
```

Se define un tipo de registro `Curso` que representa el dictado de una materia donde un docente enseña a muchos alumnos. Se crean las personas `pedro`, `maria`,

`javier` y un curso donde `javier` es el docente designado y `maria` y `pedro` son sus alumnos (representados como una lista de personas).

4.1.3. Variantes

El lenguaje de programación XGOBSTONES agrega el concepto de tipo variante, que representan la unión de varios tipos, cada uno análogo a un registro. En un tipo variante cada caso de la unión está asociado a una etiqueta, que sirve para identificarlo. Junto a esta estructura de datos, XGOBSTONES incorpora mecanismos para su definición, construcción y observación, de manera análoga a los registros, y una expresión adicional para distinguir entre los distintos casos.

La definición de un variante se logra utilizando las palabras clave `type-is-variant` y especificando cada uno de los casos del variante; cada caso puede incluir un conjunto de campos. Mostramos una definición de un variante `Gusto` que esquematiza la forma más simple de un variante, donde define casos sin campos, y luego una definición de un variante `Helado`, que define casos con sus respectivos campos.

```
type Gusto is variant
 { case Frutilla
   case Sambayon
   case Chocolate }


type Helado is variant
 {
   case Vasito
    { field bocha }

   case Cucurucho
    { field bochaArriba
      field bochaAbajo }

   case Pote
    { field gustos }
 }
```

La palabra clave `case` permite definir los casos del variante, e.g., el variante `Gusto` define los casos `Frutilla`, `Sambayon` y `Chocolate`. La palabra clave `field` permite definir los campos de cada caso del variante, tal y como se explicó para la definición de registros en la sección anterior.

La construcción de un variante se realiza a través de su constructor, análogo al constructor de registros, definido por el nombre del caso del variante y una tupla de pares campo-valor. Mostramos la construcción de cada uno de los casos definidos en el ejemplo anterior.

```
program
 {
   g1 := Frutilla
   g2 := Sambayon
   g3 := Chocolate
   h1 := Vasito(bocha <- Frutilla)
   h2 := Cucurucho(bochaArriba <- Sambayon,
                   bochaAbajo  <- Chocolate)
   h3 := Pote(gustos <- [g1, g2, g3])
   h4 := Pote(gustos <- [Chocolate, Chocolate, Sambayon])
 }
```

Junto a los variantes se introduce la expresión `match-to`, análogo de expresiones al comando de alternativa indexada, que denota una expresión distinta asociada a cada caso del variante. Mostramos la función `precioHelado` que permite calcular el precio para un helado `h` dado.

```
function precioGusto(g)
 {
   return
    (
      match (g) to              # Observar que match-to es
        Frutilla  -> 3          # una expresión y por tanto
        Sambayon  -> 6          # puede usarse en el return
        Chocolate -> 5
    )
 }

function sumGustos(gs)
 {
   suma := 0
   foreach g in gs
    { suma := suma + precioGusto(g) }
   return(suma)
 }

function precioHelado(h)
 {
   return
     (
       match (h) to
         Vasito    -> precioGusto(bocha(h))
         Cucurucho -> precioGusto(bochaArriba(h)) +
                      precioGusto(bochaAbajo(h))
         Pote      -> sumGustos(gustos(h))
     )
 }
```

Observar que `match-to` es una expresión, y por lo tanto denota un valor. Por

esa razón se puede utilizar como la expresión que denota el valor retornado por la función `precioGusto` y `precioHelado`.

4.2. Pasaje por referencia y memoria

Todas las características vistas hasta el momento son funcionalmente puras[2]. Sin embargo, es importante trabajar las nociones de memoria y de pasaje de parámetros por referencia. El lenguaje de programación XGobstones incorpora estas nociones mediante tres elementos: el operador *punto*, la definición de un parámetro de pasaje por referencia en procedimientos y punto de entrada `program` y los arreglos.

4.2.1. Operador *punto*

El lenguaje provee el operador *punto* para lograr el acceso y la modificación de los campos de una variable de registro. El siguiente ejemplo muestra la modificación de una variable de registro de tipo `Persona` mediante esta característica.

```
program
 {
   r := Persona(nombre <- "Javier",
                dni    <- "0"
                edad   <- 20)
   nombreR  := nombre(r)    # "Javier"

   r.nombre := "Ramiro"
   nombreR2 := nombre(r)    # "Ramiro"

   r.dni    := "4433353"
   dniR     := dni(r)       # "4433353"

   r.edad   := 44
   edadR    := edad(r)      # 44
 }
```

Vemos como los campos `nombre`, `dni` y `edad` de la variable de registro `r` son asignados mediante el operador *punto*, evitando la necesidad de construir un nuevo registro casi idéntico al anterior con excepción al valor del campo que se desea reemplazar. En el ejemplo puro presentado en la sección 4.1.2, tanto `r1` como `r2`, `r3` y `r4` coexisten y pueden usarse de manera independiente. En cambio, al usar el operador punto, el valor original de `r` es *reemplazado* por un nuevo valor, como si `r` se tratase de una memoria.

[2]Una operación invocada con los mismos argumentos siempre denota el mismo valor.

4.2.2. Parámetro de pasaje por referecia en procedimientos

El lenguaje incorpora pasaje de parámetros por referencia a través de la especificación de un parámetro especial en la definición de procedimientos o del punto de entrada `program`. Este parámetro especial se indica anteponiendo un nombre, separado por un punto, al nombre del procedimiento o a la palabra clave `program`. Esta característica es opcional a nivel del programa: en caso de querer utilizarla en algún punto del programa, el programador debe emplearla en todo el programa. Si no se utiliza esta característica, XGOBSTONES es retrocompatible con los programas GOBSTONES 3.0. Sin embargo, al utilizarla, el tablero se vuelve un elemento explícito, y se pierde la retrocompatibilidad. Por una decisión durante el diseño del lenguaje, los arreglos y los registros modificables sólo están disponibles cuando se utiliza esta característica. El siguiente ejemplo ilustra la utilización del pasaje de parámetros por referencia.

```
procedure x.Incrementar()
 { x := x + 1 }


t.program
  {
    n := 3
    m := -8
    n.Incrementar()
    m.Incrementar()
    return(n, m)     # n -> 4; m -> -7
  }
```

Este ejemplo define un procedimiento llamado `Incrementar` que recibe el parámetro por referencia `x` y lo incrementa en uno. Luego, define un bloque `program` que define un parámetro por referencia `t` (del cual se hablará en la sección 4.2.4) e incrementa el valor de las variables `n` y `m` utilizando el procedimiento anterior.

La presencia de esta característica, junto a la notación elegida, permite emular algunos elementos del paradigma de la programación orientada a objetos.

El siguiente ejemplo define un registro `Persona` y los procedimientos que definen parámetros por referencia `Crecer`, que incrementa la edad de una persona en uno, y `SetNombre`, que actualiza el nombre de una persona. Los parámetros por referencia pueden recibir cualquier nombre, pero en este ejemplo elegimos llamarlos `this` intencionalmente para resaltar la similtud de esta característica, empleada de este modo, con el uso de la pseudo-variable *this* (o *self*) presentes en lenguajes de programación orientada a objetos.

```
type Persona is record
 {
   field nombre
   field dni
   field edad
 }
```

```
procedure this.Crecer()
 { this.edad := this.edad + 1 }

procedure this.SetNombre(nuevoNombre)
 { this.nombre := nuevoNombre }

t.program
  {
    p := Persona(nombre <- "Juan",
                 dni    <- "20321321",
                 edad   <- 20)
    p.Crecer()
    p.SetNombre("Pedro")
    return(p)
  }
```

Este programa crea una persona con el nombre `"Juan"`, `dni "20321321"` y `edad 20` y, a partir de las modificaciones realizadas, retorna una persona con el nombre `"Pedro"`, `dni "20321321"` y `edad 21`.

4.2.3. Arreglos

El lenguaje en su versión con pasaje de parámetros por referencia incorpora arreglos, una secuencia de elementos indexada y con orden de acceso $\Theta(1)$[3]. Junto a ellos, incorpora mecanismos para su generación e indización. Los arreglos pueden verse como una agrupación en memoria de variables del mismo tipo, accesibles a través de un índice.

La creación de un arreglo se logra a través de la utilización de la expresión `Arreglo(size <- n)` que crea un arreglo de longitúd `n`. El acceso a cada uno de los elementos del arreglo se da a través de la indización. La inicialización de esta estructura se realiza a mano asignando cada uno de sus campos. Además es posible conocer la longitud de un arreglo a través de la función `size`. El siguiente ejemplo muestra la utilización del constructor de arreglos, la función `size` y la asignación de elementos del arreglo en un programa que construye un arreglo con diez elementos y lo inicializa con números del uno al diez.

```
program
 {
   a := Arreglo(size <- 10)
   foreach i in [1..size(a)]
    { a[i] := i }
 }
```

Observar la notación clásica mediante corchete para la indización del arreglo.

[3]Notación asintótica Θ [8]. Establece el peor caso del tiempo de ejecución teórico de un algoritmo.

4.2.4. Tablero como valor

Al utilizar la característica de pasaje de parámetros por referencia, el punto de entrada `program` debe indicar una variable destacada. Dicha variable, llamada `t` en los ejemplos de la sección anterior, comienza con el valor del tablero inicial, incorporando así el tablero como un valor más, modificable únicamente a través del mecanismo de pasaje por referencia. A este modo lo llamamos de *tablero explícito* y el mismo utiliza un conjunto distinto de comandos y expresiones para interactuar con el tablero (con respecto a los utilizados en la sección 2.1). Los comandos esperan un tablero pasado por referencia mientras que las expresiones requieren de un tablero como primer argumento.

```
t.program
  {
    t.Mover(Este)
    t.Poner(Rojo)
    if (hayBolitas(t, Rojo))
     { t.Poner(Verde) }
  }
```

Al incorporar al tablero como un valor más dentro de XGOBSTONES se posibilita el hecho de tener varias copias del tablero, manipulando cada instancia independientemente. En la figura 4.2.1 se presenta un ejemplo de este comportamiento.

5. El ambiente de desarrollo PYGOBSTONES 1.0

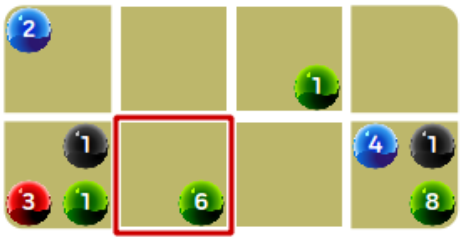
Como establecimos en las secciones anteriores, el objetivo de este trabajo es contribuir al desarrollo de la herramienta PYGOBSTONES 1.0 que implementa un ambiente de trabajo para los lenguajes de programación GOBSTONES 3.0 y XGOBSTONES.

Esta nueva herramienta, implementada en PYTHON [6], está compuesta por tres partes: el compilador y máquina virtual del lenguaje de programación GOBSTONES 3.0, el compilador y máquina virtual del lenguaje de programación XGOBSTONES y la interfaz gráfica.

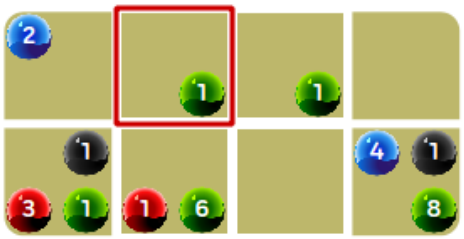
Decidí desarrollar un compilador y una máquina virtual independientes para cada lenguaje implementado debido al corto tiempo de vida de XGOBSTONES y a la incertidumbre respecto de su futuro al momento de comenzada esta tesina. El lenguaje GOBSTONES, por otro lado, constituye un lenguaje maduro que ya cuenta con tres versiones y que es probable que su diseño se mantenga estable.

En esta sección describo la implementación del compilador y máquina virtual de GOBSTONES 3.0 a partir de los existentes de GOBSTONES 2.0 (incluido en PYGOBSTONES 0.97), la implementación del compilador y máquina virtual de XGOBSTONES a partir de los desarrollados para GOBSTONES 3.0 y el desarrollo de un módulo de comunicación que permita la interacción entre la interfaz gráfica y los compiladores y máquinas virtuales.

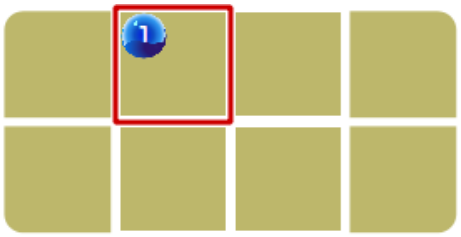
```
t.program
  {
    t.IrAlBorde(Sur)
    t.IrAlBorde(Oeste)
    t.Mover(Este)

    t1 := t
    t1.Poner(Rojo)
    t1.Mover(Norte)
    t1.Poner(Verde)

    t2 := t1
    t2.VaciarTablero()
    t2.Poner(Azul)
  }
```



Tablero `t`



Tablero `t1`



Tablero `t2`

Figura 4.2.1: A la izquierda un programa que realiza sucesivas copias del tablero aplicando diferentes operaciones. A la derecha se muestran los tableros finales `t`, `t1` y `t2` al terminar (si bien solo el tablero denotado por `t` se puede visualizar en la herramienta).

Para la presentación de las implementaciones hago uso de dos herramientas auxiliares que se explican a continuación.

5.1. Herramientas auxiliares

Junto a las implementaciones del lenguaje desarrollé una herramienta para la generación de casos de prueba automáticos, descripta en la sección 5.1.1, y una notación para generalizar ejemplos, principalmente durante la descripción de los procesos de compilación, descripta en la sección 5.1.2.

5.1.1. Herramienta para los casos de prueba

Con el objetivo de verificar el correcto funcionamiento de las características implementadas, desarrollé algunas herramientas para facilitar la creación de casos de prueba [3]. Con ellas, definí una batería de casos de prueba tanto para programas GOBSTONES 3.0 como XGOBSTONES. Por simplicidad, en esta sección hablaré solo de la generación de casos de prueba para GOBSTONES 3.0 puesto que los de XGOBSTONES siguen el mismo principio de trabajo.

En primer lugar, creé una herramienta que permite la generación de casos de prueba automáticos a partir de un programa GOBSTONES, un programa equivalente en PYTHON y un conjunto de datos de prueba. Con estos elementos la herramienta genera casos de prueba que verifican que la ejecución ambos programas obtengan el mismo resultado a partir del mismo subconjunto de datos.

Al presentar los casos de prueba utilizados, daré un esquema de programa GOBSTONES 3.0 y un programa PYTHON que serán los datos usados por la herramienta para generar ese caso de prueba. El esquema de programa en GOBSTONES 3.0 usará atributos externos cuyos valores serán provistos por la herramienta durante la ejecución de los tests. Estos atributos se denotan mediante una *anotación*, un identificador precedido por un arroba (e.g. `@nombre`). Los datos externos empleados en el esquema de programa GOBSTONES 3.0 se encuentran disponibles para el programa PYTHON a través del diccionario llamado `args` (e.g. `args["nombre"]`).

Para la utilización de la herramienta se define una subclase de la clase `TestScript` donde se incluyen los programas provistos. La clase `TestScript` posee el comportamiento básico para la integración de un test a la herramienta. Por ejemplo, para testear la suma y la resta de GOBSTONES 3.0 usaré el siguiente par de programas:

```
# Esquema de programa          # Programa Python
# Gobstones 3.0                return (args["a"] + args["b"],
return (@a + @b, @a - @b)              args["a"] - args["b"])
```

que serán parte del siguiente código PYTHON dentro de la clase `OperadorTest` que implementa este test.

```
class OperatorTest(TestScript):
    def __init__(self):
        super(OperatorTest, self).__init__({
            "a": [3, 0, 10, 3000, 99999],
            "b": [8, 0, 33, 5213, 56655]
        })


    def nretvals(self):
        return 2


    def gbs_code(self):
        return """
            # Esquema de programa
            # Gobstones 3.0
            return (@a + @b, @a - @b)
        """


    def pyresult(self, args):
        # Program Python
        return (args["a"] + args["b"],
                args["a"] - args["b"])
```

Esta clase define las funciones `gbs_code` y `pyresult` que implementan los programas mostrados en cada ejemplo y las operaciones de inicialización. En este ejemplo, la anotación `@a` variará en el conjunto $\{3, 0, 10, 3000, 99999\}$ y la anotación `@b` en el conjunto $\{8, 0, 33, 5213, 56655\}$.

La herramienta, por otro lado, también permite la ejecución de programas GOBSTONES 3.0 o XGOBSTONES escritos manualmente que serán utilizados como casos de prueba. Estos programas deberán retornar las variables nombradas `passed` y `failed` para indicar los tests que resultaron exitosos y los que fallaron respectivamente, o, al declarar la anotación `#!assert` como primera línea de programa, una tupla de valores de verdad donde cada uno representa el resultado de un test individual. A continuación se presenta un ejemplo de un programa GOBSTONES 3.0 que verifica que el generador de rangos y el generador de secuencias explícitas posean el mismo comportamiento.

```
#!assert

function sumaDesplazada(xs)
 {
   sum := 0
   foreach x in xs
    { sum := sum * 10 + x }
   return(sum)
 }
```

```
program
 {
  sumA := sumaDesplazada([1,2,3,4,5,6,7,8,9,10])
  sumB := sumaDesplazada([1..10])
  return(sumA == sumB)
 }
```

Estas herramientas para la generación de casos de prueba se incluyen dentro de las implementaciones de los lenguajes.

5.1.2. Notación específica

Con el fin de lograr que los ejemplos dados resulten genéricos, utilizaré una notación específica que establece una forma de nombrar fragmentos de texto que varía y que permite reutilizarlos en otro punto del ejemplo.

Esta notación se emplea utilizando un nombre entre los símbolos `<` y `>` (e.g. `<Nombre>`), para identificar una porción de código que varía y su aparición estará acompañada por una explicación de su significado. También usaré la notación `<Nombre-i>` para identificar elementos del programa que comparten la misma naturaleza donde `i` representa una etiqueta numérica, alfabética o una palabra completa. Por ejemplo, el texto `[<Expr-1>, <Expr-2>, <Expr-3>]` contiene tres expresiones particulares, a ellas me referiré tanto por su nombre particular; e.g. `<Expr-3>`, como por su generalización, `<Expr-i>`. A continuación se muestra un fragmento de código GOBSTONES 3.0 que posee estas anotaciones.

```
if (<Condition>)
  then <Block-Then>
  else <Block-Else>
```

Este fragmento posee las anotaciones `<Condition>`, que representa una expresión de tipo booleano, y `<Block-i>` que representa un bloque de comandos.

Estas anotaciones son utilizadas, entre otros usos, para indicar en el bytecode la aparición del elemento anotado ya sea de manera literal o con un texto que describe alguna modificación. Por ejemplo, la presencia de la anotación `<Bytecode de <Nombre>>` representa el bytecode resultante de la compilación del fragmento de código representado por la anotación `<Nombre>`.

Por otra parte, se utilizarán anotaciones para reemplazar las etiquetas empleadas junto a la instrucción `label`: dado que las etiquetas utilizadas generadas por el compilador son simples números aleatorios (e.g. `L_191290533`), decidimos reemplazarlas por anotaciones cuyo nombre favorezcan la comprensión de los ejemplos.

El siguiente bytecode muestra la compilación del fragmento de código anterior utilizando las anotaciones presentes en él.

```
<Bytecode de <Condition>>
jumpIfFalse <L_ELSE>
<Bytecode de <Block-Then>>
jump <L_END>
label <L_ELSE>
<Bytecode de <Block-Else>>
label <L_END>
```

En este ejemplo, las anotaciones `<L_ELSE>` y `<L_END>` representan etiquetas generadas aleatoriamente por la máquina virtual. Notar que estas etiquetas varían para cada alternativa compilada, pero son la misma en cada uso dentro del mismo fragmento.

Para entender el uso de los *jumps* en el ejemplo es necesario tener en cuenta que la compilación de una expresión deja el resultado en la pila de ejecución y su compilación dependerá de si se trata de una constante (`pushConst`), una variable (`pushVar`) o invocación de una función (compilación de los argumentos y luego una instrucción `call` invocando la función).

También proveeré, en ocasiones, comentarios que ayuden a comprender el resultado de la compilación, posicionando cada línea del código compilado en la primer línea del bytecode al cual se tradujo, que a su vez hacen uso de estas anotaciones.

```
<Bytecode de <Condition>>     # if (<Condition>)
jumpIfFalse <L_ELSE>          #
<Bytecode de <Block-Then>>    #   then <Block-Then>
jump <L_END>                  #
label <L_ELSE>                #   else
<Bytecode de <Block-Else>>    #     <Block-Else>
label <L_END>                 #
```

A continuación de listan una serie de anotaciones de uso frecuente en los ejemplos que se presentarán.

`<Expr>` denota una expresión cualquiera,

`<Condition>` denota una expresión booleana,

`<List>` denota una lista cualquiera,

`<Block>` denota un bloque de comandos,

`<Index>` denota un nombre que es utilizado como índice,

`<L_tag>` denota una etiqueta aleatoria generada por el compilador, donde `tag` será reemplazado por un nombre que aporte significación al ejemplo. Por ejemplo, la anotación `<L_BEGIN_WHILE>` representa una etiqueta que se utiliza para indicar el principio de un `while`, que varía para cada `while` compilado.

5.2. Implementación de GOBSTONES 3.0

Para la implementación del compilador y la máquina virtual de GOBSTONES 3.0 extendí los existentes de GOBSTONES 2.0 que forman parte de la herramienta PYGOBSTONES 0.97. Fue necesario cambiar la implementación de cada una de las etapas del compilador y también la máquina virtual para incorporar las nuevas características del lenguaje. Los cambios a las etapas de análisis semántico y compilación requirieron modificar el código PYTHON existente y agregar nuevas operaciones.

Además, implementé casos de prueba para cada una de las características desarrolladas, utilizando la herramienta mencionada en la sección 5.1.1.

5.2.1. Cambios en el bytecode de GOBSTONES

Para poder implementar las nuevas características a GOBSTONES 3.0 se introdujeron algunos cambios en el conjunto de instrucciones del bytecode. Para explicitar la utilización de la pila de programa, la instrucción `assign` se renombró a `popTo`. Con el mismo concepto la instrucción `pushVar` se renombró a `pushFrom`. Otros cambios incluyeron el renombre de la instrucción `BOOM` a `THROW_ERROR`, la incorporación de la instrucción `entrypoint` (ver sección 5.2.3) y la incorporación de las instrucciones `setImmutable` y `unsetImmutable`, de efectos opuestos, que establecen si una variable es inmutable o no lo es.

La gramática detalla del nuevo bytecode se presenta en el apéndice E.

5.2.2. Cambios de sintaxis

En cada ejecución, el compilador, en su etapa de análisis sintáctico, hace uso de una gramática BNF que utilizará para analizar sintácticamente el programa ingresado como entrada (como se explicó en la sección 2.3.1).

Implementé los cambios de sintaxis que incorpora el lenguaje modificando la gramática BNF de GOBSTONES 2.0 utilizada en la etapa de análisis sintáctico de PYGOBSTONES 0.97. En el apéndice G.1 se muestran los fragmentos de código que resultan significativos, aprovechando el espacio de esta sección para mostrar ejemplos.

La incorporación del nuevo punto de entrada `program` la realicé modificando la regla que define una definición del lenguaje agregando este elemento. De la misma forma, agregué la sintaxis de la repetición simple a la gramática. Para la adaptación de la nueva sintaxis del comando `if-then-else` bastó con agregar un símbolo terminal opcional `then`. La modificación de la alternativa indexada se logró cambiando los símbolos terminales `case-of` por `switch-to`. La incorporación de los comentarios al estilo PYTHON debía ser un trabajo trivial (dado que el la gramática BNF de GOBSTONES 2.0 ya contemplaba la presencia de comentarios de línea y multi-línea); sin embargo el símbolo `#` utilizado para los comentarios de línea entraba en conflicto con el símbolo utilizado para comentar líneas de la gramática BNF, por lo que fue necesario modificar la etapa de análisis sintáctico para interpretar el símbolo `##`, ahora presente en la gramática,

como nuevo comentario de línea. En el apéndice G.1.4 mostramos las gramática para los comentarios en sus dos versiones.

Por último la repetición indexada fue uno de los elementos sobre los cuales tuvieron mayor impacto los cambios que introdujo GOBSTONES 3.0. Para implementar los cambios sintácticos de la repetición indexada comencé por cambiar la palabra clave `repeatWith` por `foreach` y luego crear la sintaxis adecuada que permita utilizar secuencias explícitas, además de rangos.

Para verificar la nueva sintaxis de estas características utilice el siguiente caso de prueba que resulta exitoso si el mismo es ejecutado y termina sin errores.

```
#!assert
# Comentario de línea
""" Comentario
    multi-línea """
program
 {
   x := True

   if (x)
   then { Skip }
   else { Skip }

   switch (x) to
     _ -> { Skip }

   foreach i in [1,2,3,4,5] { Skip }
   foreach i in [1..5] { Skip }
   foreach i in [1,3..5] { Skip }
   repeat (3) { Skip }

   return(True)
 }
```

5.2.3. El punto de entrada `program`

Para la implementación del punto de entrada `program` incorporé al compilador y máquina virtual un nuevo concepto que se suma a la definición de procedimientos y funciones: los *entrypoint* (punto de entrada). De esta manera, es posible en un futuro agregar a partir de este concepto nuevos puntos de entrada que involucren diferentes comportamientos del compilador y máquina virtual, como podría ser `interactive program`.

Para esta implementación fue necesario modificar, además de la etapa de análisis sintáctico de la que se habló en la sección 5.2.2, la etapa de análisis semántico y la etapa de compilación.

En la etapa de análisis semántico implementé verificaciones sobre la estructura del programa GOBSTONES 3.0 en general y la estructura del punto de entrada `program` en particular. Estas verificaciones son:

- El programa debe tener un punto de entrada `program` y no más que uno.
- El punto de entrada `program` puede incluir en su bloque un comando `return`. Si lo hace debe ser el último comando del bloque.

Finalmente en la etapa de compilación, se compila el punto de entrada `program` a bytecode de manera similar a la compilación de una función o procedimiento. El primer paso fue extender el conjunto de instrucciones con la nueva instrucción `entrypoint` que indica el comienzo del programa. Al traducir el `program` a bytecode, se agrega la instrucción `entrypoint`, se compila el bloque y se agrega una instrucción que indica el fin de la definición. Dentro del bloque de `program` se compila el comando `return` (si no forma parte del bloque se agrega este comando sin argumentos) como una instrucción ya existente `returnVars` que constituye un caso especial de la instrucción `return` que indica el final de la ejecución y, opcionalmente, expresiones que deben formar parte del resultado final.

El bytecode generado para un punto de entrada `program` se muestra a continuación.

```
# Estructura del punto de entrada program
program
  <Block>

# Compilación del punto de entrada program
entrypoint $program        # program
  <Bytecode de <Block>>    #   <Block>
end                        #
```

La verificación de esta característica se realiza en conjunción con los restantes casos de prueba puesto que es parte de todos ellos.

5.2.4. Repetición simple

La implementación de la repetición simple se realiza simulando la misma como si se tratase de un while con un contador, reutilizando por tanto la generación de código preexistente. Se muestra el bytecode resultado de la compilación de esta estructura con comentarios que clarifican su funcionamiento.

```
# Estructura de la repetición simple
repeat <Expr> <Block>

# Compilación de la repetición simple
<Bytecode de <Expr>>       # counter := <Expr>
popTo _counter             #
label <L_BEGIN_WHILE>      # while (true) {
pushFrom _counter          #     if (not (counter > 0))
pushConst 0                #
call > 2                   #
jumpIfFalse <L_END_REPEAT> #         { break }
```

```
<Bytecode de <Block>>      #     <Block>
pushFrom _counter          #     counter := counter -1
pushConst 1                #
call - 2                   #
popTo _counter             #
jump <L_BEGIN_WHILE>       # }
label <L_END_REPEAT>       #
```

El bytecode resultante muestra la incialización de un contador con el valor denotado por `<Expr>` y bucle implementado mediante etiquetas e instrucciones de salto que ejecutará las instrucciones del bloque y decrementará el contador en cada iteración. Este bucle terminará cuando el valor del contador denote un número menor o igual a cero.

Verifico esta característica a partir de los programas que muestro a continuación y un conjunto de números enteros aleatorios generados automáticamente.

```
# Esquema de programa
# Gobstones 3.0
count := 0
repeat (@times)
 { count := count +1 }
return(count)
```

```
# Program Python
return args["times"]
```

El programa GOBSTONES 3.0 realiza un conteo de las iteraciones de la repetición simple. El programa PYTHON simplemente indica el resultado esperado para el conteo.

5.2.5. Repetición indexada

La implementación de la repetición indexada presente en el compilador de GOBSTONES 2.0 es específica para la forma de rangos simples. Para la implementación del nuevo comporamiento de la repetición indexada de GOBSTONES 3.0 utilicé la misma implementación que la definida para XGOBSTONES, que representa internamente los rangos y las secuencias explícitas como listas (ver sección 5.3.2).

Esto completa los cambios para llevar el comportamiento de GOBSTONES 2.0 a GOBSTONES 3.0.

5.3. Implementación de XGOBSTONES

Desarrollé el compilador y la máquina virtual de XGOBSTONES usando como punto de partida las implementaciones de GOBSTONES 3.0 explicadas en la sección anterior.

5.3.1. El bytecode y la máquina virtual de XGOBSTONES

El bytecode utilizado por XGOBSTONES es el mismo que el presentado en la sección 5.2.1 para GOBSTONES 3.0. Sobre éste se construyen los procesos de

compilación de los nuevas características incorporadas en este lenguaje.

Para la implementación de la máquina virtual de XGOBSTONES extendí la desarrollada para GOBSTONES 3.0 e incorporé un conjunto de operaciones internas, inaccesibles al programador, a partir las cuales implementé las estructuras de datos y el mecanismo de pasaje por referencia. En este conjunto de operaciones se encuentran `_range`, `_construct`, `_construct_from`, `_mk_field`, `_extract_case`, `_getRef`, `_getRefValue`, `_SetRefValue`, `mk_array`, etc. Cada una estas operaciones se explicarán conforme se hagan presentes en las secciones siguientes.

5.3.2. Listas

Para representar las listas de XGOBSTONES elegí usar el tipo primitivo `list` de PYTHON. Dado que la máquina virtual opera con valores PYTHON, esto permite que las listas puedan ser manipuladas por la misma sin mayores cambios.

Comencé por implementar los tres generadores básicos: lista nula `[]`, lista singular `[x]` y concatenación `xs ++ ys`, internamente defindos como funciones provistas por la máquina virtual que operan sobre la representación en PYTHON. Las listas primitivas de PYTHON proveen `[]` y `[x]` como notación y el operador `+` para la concatenación.

Implementé las secuencias explícitas a partir de los generadores básicos: en la etapa de análisis sintáctico traduzco las secuencias explícitas en una serie de concatenaciones de listas singulares. Por ejemplo, la secuencia explícitas `[1,2,3]` se traduce internamente en la expresión `[1] ++ [2] ++ [3] ++ []`.

Implementé los rangos mediante una primitiva de la máquina virtual, `_range`, que a partir del primer y último elemento del rango y, opcionalmente, el segundo elemento, retorna una lista con los elementos correspondientes. Por ejemplo, el rango `[1,7..31]` se traduce internamente en la expresión `_range(1, 31, 7)` que retorna la lista `[1, 7, 13, 19, 25, 31]`.

Extendí el conjunto de primitivas de la máquina virtual para incluir operaciones que representen `head`, `tail`, `init`, `last` e `isEmpty`. La máquina virtual ejecuta estas primitivas usando operaciones de PYTHON de seccionado de listas y la función `len`. Por ejemplo, `tail(xs)` se representa con `xs[1:]`, `last(xs)` con `xs[-1]` e `isEmpty(xs)` con `len(xs) == 0`.

La implementación del nuevo comportamiento de la repetición indexada se redujo a modificar la compilación a Bytecode. La nueva representación de la repetición indexada es básicamente un recorrido sobre la lista utilizando las expresiones `head`, `tail` e `isEmpty`.

```
# Estructura de la repetición indexada
foreach <Index> in <List> <Block>


# Compilación de la repetición indexada
<Bytecode de <List>>      # xs0 := <List>
popTo _xs0                #
label <L_WHILE_BEGIN>     # while (true) {
```

```
pushFrom _xs0                #     if (isEmpty(xs0))
call isEmpty 1               #
call not 1                   #
jumpIfFalse <L_WHILE_END>    #       { break }
pushFrom _xs0                #     <Index> := head(xs0)
call head 1                  #
popTo <Index>                #
setImmutable <Index>         #     setImmutable(<Index>)
<Bytecode de <Block>>        #     <Block>
unsetImmutable <Index>       #     unsetImmutable(<Index>)
pushFrom _xs0                #     xs0 := tail(xs0)
call tail 1                  #
popTo _xs0                   #
jump <L_WHILE_BEGIN>         #
label <L_WHILE_END>          # }
```

La variable `xs0` mantiene el estado de la lista que se irá reduciendo conforme a las iteraciones del bucle. En cada iteración se asigna la cabeza de la lista `xs0` a la variable `<Index>`. La ejecución de este código concluye cuando la lista `xs0` toma el valor `[]`. Se utiliza la instrucción `setImmutable` para que la variable `<Index>` se vuelva inmutable durante la ejecución de `<Block>` y luego la instrucción `unsetImmutable` para deshacer la restricción anterior.

La verificación del generador de rango se realiza a mediante la comparación del resultado de los programas que se muestran a continuación variando la etiqueta `@low` en el conjunto $\{1, 11, 33\}$ y la etiqueta `@high` en el conjunto $\{11, 22, 51\}$.

```
# Esquema de programa
# XGobstones

xs := [@low..@high]
ys := [@low, @low+1..@high]
zs := [@high,@high-1..@low]
ws := [@low, @low+5..@high]
vs := [@high,@high-5..@low]
us := [@low, @low+9..@high]
ts := [@high,@high-9..@low]

return(ts, us, vs, ws, xs,
       ys, zs)
```

```
# Programa Python

low  = args['low']
high = args['high']
xs = range(low,  high+1)
ys = range(low,  high+1, 1)
zs = range(high, low-1, -1)
ws = range(low,  high+1, 5)
vs = range(high, low-1, -5)
us = range(low,  high+1, 9)
ts = range(high, low-1, -9)

return ts, us, vs, ws, xs,
       ys, zs
```

Las verificaciones para las secuencias explícitas y el comportamiento de la nueva implementación de la repetición indexada están dadas por el caso de prueba descripto por los programas que se muestran a continuación, donde `@list` es una secuencia explícita de números generada aleatoriamente (e.g. `[8,10,33,84,11]`).

```
# Esquema de programa
# XGobstones

res := 0
foreach n in @numbers
 { res := res*10 + n }

return(sum)
```

```
# Programa Python

res = 0
ns = ns["numbers"][1:-1]
ns = ns.split(",")
for n in map(int, ns):
    res = res*10 + n
return res
```

Esta verificación realiza una sumatoria de todos los números de la lista realizando desplazando el resultado en cada suma, con lo que el resultado depende del orden de los elementos de la lista.

5.3.3. Registros

Implementé los registros de XGOBSTONES utilizando como representación los diccionarios provistos por PYTHON, donde cada campo se corresponde con un índice del diccionario.

Para la construcción de un registro, la expresión de XGOBSTONES se compila como si hubiese sido escrita usando operaciones internas de la máquina virtual `_construct` y `_mk_field` como se muestra en el siguiente ejemplo.

```
# Código XGobstones
p:= Persona(nombre <- "Pedro",
            dni    <- "2132132",
            edad   <-  34)


# Código a compilar
p := _construct("Persona",
                [_mk_field("nombre", "Pedro"),
                 _mk_field("dni", "2132132" ),
                 _mk_field("edad", 34)])
```

La función `_mk_field` asocia un nombre de campo con un valor, y la función `_construct` construye la representación PYTHON a partir de un constructor dado y una lista de asociaciones (campo, valor) generadas a partir de `_mk_field`.

Para el caso de la construcción de un registro a partir de un registro existente se hace uso de la función `_construct_from`, similar a `_construct`, que recibe como tercer argumento una expresión de tipo registro sobre la cual se construirá el nuevo registro.

```
# Código XGobstones
p:= Persona(nombre <- "Pedro",
            dni    <- "2132132",
            edad   <-  34)

q := Persona(p | dni <- "1233323")
```

```
# Código a compilar
p := _construct("Persona",
                [_mk_field("nombre", "Pedro"),
                 _mk_field("dni", "2132132" ),
                 _mk_field("edad", 34)])

q := _construct_from("Persona",
                     [_mk_field("dni", "1233323")],
                     p)
```

La implementación de las funciones observadoras de campo consiste en su traducción en tiempo de compilación en llamadas a la función `_get_field`, operación interna de la máquina virtual, que recibe un registro y un nombre de campo y devuelve el valor de dicho campo.

```
# Código XGobstones
d := dni(p)

# Código a compilar
d := _get_field(p, "dni")
```

5.3.4. Variantes

Para la representación de los variantes usé el mismo enfoque que utilicé para representar los registros: cada caso del variante se representa utilizando un diccionario PYTHON. Para la construcción de un variante, al igual que en la construcción de un registro, se utiliza la función `_construct` que recibe el constructor del caso y un listado de tuplas `(campo, valor)` creadas utilizando la función `_mk_field`.

```
type Gusto is variant
  {
    case Frutilla
    case Chocolate { field amargo }
  }

# Código XGobstones
g1 := Frutilla
g2 := Chocolate(amargo <- True)

# Código a compilar
g1 := _construct("Frutilla", [])
g2 := _construct("Chocolate", [_mk_field("amargo", True)])
```

Implementé la expresión `match-to` como una secuencia de alternativas que buscan coincidir el caso del argumento con alguno de los casos definidos. Si no se cubren todos los casos del variante, la etapa de análisis semántico exige la definición de un caso por defecto. Para la extracción del caso del argumento se

utiliza la operación interna _extract_case, que, dado un valor de tipo variante, retorna el constructor del caso con el que fue construido dicho valor.

```
# Estructura de la expresión match-to

match (<Expr-V>) of
    <Case-1> -> <Expr-1>
    <Case-2> -> <Expr-2>
    ...
    <Case-N> -> <Expr-N>
    _        -> <Expr-Else>

# Compilación del match-to

<Bytecode de <Expr-V>>      # case := _extract_case (
call _extract_case 1        #           <Expr-V>
popTo _case                 #         )

pushFrom _case              # if (type == <Case-1>)
pushConst <Case-1>          #
call == 2                   #
jumpIfFalse  <L_CASE_2>     #
<Bytecode de <Expr-1>>      #   { push(<Expr-1>) }
jump <L_MATCH_END>          #

label <L_CASE_2>            #
pushFrom _case              # elif (type == <Case-2>)
pushConst <Case-2>          #
call == 2                   #
jumpIfFalse  <L_CASE_3>     #
<Bytecode de <Expr-2>>      #   { push(<Expr-2>) }
jump <L_MATCH_END>          #

...                         # ...

label <L_CASE_N>            #
pushFrom _case              # elif (type == <Case-N>)
pushConst <Case-N>          #
call == 2                   #
jumpIfFalse  <L_ELSE>       #
<Bytecode de <Expr-N>>      #   { push(<Expr-N>) }
jump <L_MATCH_END>          #

label <L_ELSE>              # else {
<Bytecode de <Expr-Else>>   #   push(<Expr-Else>)
label <L_MATCH_END>         #  }
```

La alternativa indexada se implementa a partir de una secuencia de saltos condicionales que verifican que el constructor de caso utilizado para construir la ex-

presión recibida como argumento (`<Expr-V>`), almacenado en la variable `case`, coincida con alguno de los literales que describen casos del variante (`<Case-i>`). Ante una coincidencia se apila la expresión asociada con el literal (`<Expr-i>`); de lo contrario se apila la expresión por defecto (`<Expr-Else>`).

5.3.5. Pasaje por referencia

Definí tres operaciones internas de la máquina virtual con las cuales expresé la implementación de referencias: `_getRef` que, dado un *l-value* y un índice o campo, retorna una referencia a dicho valor, `_SetRefValue` que, dada una referencia y un valor, reescribe el valor referenciado, y `_getRefValue` que, dada una referencia, retorna el valor referenciado. Las operaciones mencionadas manipulan instancias de la clase `GbsRef` que representan las distintas referencias. Esta clase posee un método para la lectura del valor de la referencia (`get`) y otro para reescritura del valor referenciado (`set`). Dado que en el contexto del lenguaje de programación PYTHON no existe el concepto de pasaje por referencia, creé la clase `GbsObject` cuyas instancias ofician de *value holder*[4] para los valores de XGOBSTONES. De esta manera es posible modificar el valor contenido en una instancia de esta clase y que el cambio se vea reflejado en todo el programa. A partir de las primitivas y clases mencionadas implementé el comportamiento del operador *punto* y el mecanismo de indización para arreglos.

La implementación del operador `punto` varía dependiendo del contexto en que se lo utilice. En el contexto de una expresión lo implementé como una sucesión de llamadas a la función `_getRef` seguida de un llamado a la función `_getRefValue` que, en conjunto, devuelven el valor final de la secuencia de aplicaciones de los operadores *punto*. En cambio, en el contexto de una asignación, la sucesión de llamadas a `_getRef` se vé seguida de una llamada al procedimiento `_SetRefValue` que, a partir de la referencia obtenida y una expresión a asignar, reemplaza al valor referenciado.

El siguiente es un ejemplo que muestra la utilización de las funciones `_getRef` y `_getRefValue` para obtener el valor de un campo. La variable `var` corresponde a una variable de tipo registro, el campo `f1` es de tipo registro conteniendo un campo `f2`.

```
# Código XGobstones
fieldValue := var.f1.f2

# Código a compilar
fieldValue :=
    _getRefValue(_getRef(_getRef(var, "f1"), "f2"))
```

Notar que cada operador *punto* se traduce como una función `_getRef` que genera una referencia al campo. La función `_getRefValue` obtiene el valor efectivo.

Para ilustrar el mecanismo de asignación de una referencia, el ejemplo a continuación muestra la asignación del campo `f1` de la variable de tipo registro

[4] Patrón de diseño utilizado en lenguajes orientados a objetos para guardar referencia a un valor que es modificado.

`var` con el valor `3`.

```
# Código XGobstones
var.f1.f2 := 3

# Código a compilar
_SetRefValue(_getRef(_getRef(var, "f1"), "f2"), 3)
```

A partir de una referencia generada para el campo `f1` se genera una segunda referencia para el campo `f2` cuyo valor referenciado es reemplazado por el valor `3` mediante el procedimiento `_SetRefValue`.

5.3.6. Arreglos

Implementé los arreglos de XGOBSTONES a partir de las listas de PYTHON[5].

El constructor de arreglos lo implemento como la operación interna de la máquina virtual `_mk_array` que dado un parámetro que indica longitud crea una lista de PYTHON de dicho tamaño.

```
# Programa XGobstones
arr := Arreglo(size <- 30)

# Código a compilar
arr := _mk_array(30)
```

La expresión `size`, que retorna la longitud del arreglo, se implementa a partir de la función `len` provista por PYTHON, que retorna la longitud de la representación del arreglo.

El mecanismo de indizado funciona de la misma manera que el operador *punto* con la diferencia que se realiza una única llamada a `_getRef` con el número de índice. El siguiente es un ejemplo que muestra la asignación del valor del primer elemento de un arreglo `a` a una variable `v` y la asignación del quinto elemento del mismo arreglo con el valor `10`

```
# Código XGobstones
  v := a[1]
  a[5] := 10

# Código a compilar
v := _getRefValue(_getRef(a, 1))
_SetRefValue(_getRef(a, 5), 10)
```

Esta característica es verificada por el siguiente caso de prueba. La anotación `@size` varía en el intervalo $[1, 30]$.

[5]En la implementación de las listas de PYTHON tanto el acceso como la modificación de un elemento de la lista mediante la utilización de un índice se realiza en tiempo constante, es decir, en orden $\Theta(1)$.

```
# Esquema de programa
# XGobstones

arr := Arreglo(size <- @size)

foreach i in [1..@size]
 { arr[i] := i }

arr2 := Arreglo(size <- @size)

foreach i in [@size, @size-1..1]
 { arr2[i] := arr[i] }

return(arr2)
```

```
# Programa Python

return range(size, 0, -1)
```

Este caso de prueba crea un arreglo de longitud `@size` donde cada posición es inicializada con su índice y luego se copia a un segundo arreglo en orden inverso.

5.3.7. Chequeo de tipos dinámico

La especificación de Gobstones 3.0 y XGobstones establece que una misma variable no puede ser asignada con valores de distintos tipos, y además, en el caso de XGobstones, todos los elementos de una lista deben pertenecer al mismo tipo. Para implementar esta característica, la máquina virtual implementa el chequeo dinámico de tipos para cada operación que lo precisa mediante la utilización de la operación interna `poly_typeof`, heredada de la implementación de la máquina virtual de PyGobstones 0.97, que devuelve el nombre del tipo de un valor dado.

Para la realización de chequeo de tipos durante la asignación modifiqué la implementación de la instrucción `popTo` presente en el bytecode. La nueva implementación verifica, en caso de que la variable a asignar ya esté definida, que el tipo del nuevo valor de la variable coincida con el tipo del valor viejo y, en el caso de las listas, que el tipo interno también coincida.

Para el caso del chequeo de tipos para valores de tipo lista, implementé en el operador de concatenación `++` una verificación que asegure que los tipos internos de las listas a concatenar coincidan.

De manera similar se trata el chequeo de tipos sobre otras construcciones como los campos, los registros, etc.

5.3.8. Tablero como valor

En XGobstones existen dos modos de utilización del tablero: el modo explícito y el modo implícito. En el modo explícito es posible utilizar el tablero como un valor siendo suceptible de ser asignado a una variable y utilizado como una expresión en cualquier punto del programa. El modo implícito corresponde a la forma de manejar tableros de Gobstones 3.0 donde la existencia del

tablero es implícita para el programador. Desarrollé estos dos modos de utilización del tablero proveyendo primitivas distintas para cada uno que manipulan una misma representación del tablero.

La máquina virtual modela al tablero como un objeto de la clase `Board` que provee la siguiente interfaz para su manipulación:

- `go_to_boundary` que implementa el comportamiento de `IrAlBorde`.
- `clear_board` que implementa `VaciarTablero`.
- `put_stone` que implementa `Poner`.
- `take_stone` que implementa `Sacar`.
- `move` que implementa `Mover`.
- `num_stones` que implementa `nroBolitas`.
- `exist_stones` que implementa `hayBolitas`.
- `can_move` que implementa `puedeMover`.

Ambos modos de utilización del tablero hacen uso de esta interfaz accediéndola a través de dos conjuntos distintos de primitivas de la máquina virtual.

El modo implícito utiliza el mismo conjunto de primitivas que las disponibles en GOBSTONES 3.0 siempre haciendo referencia a un único tablero global que solo es copiado dentro del contexto de una función para que la misma no genere efectos sobre el tablero original.

El modo explícito utiliza un conjunto de primitivas distinto, con el mismo nombre pero que reciben al tablero como primer argumento: en el caso de los procedimientos el pasaje del tablero se realiza por referencia, mientras que en el caso de las funciones se realiza por valor.

Al momento de comenzar la ejecución utilizando el modo explícito, la máquina virtual envuelve el tablero en un objeto `GbsObject` y lo asocia con la variable de pasaje por referencia del punto de entrada `program`. A partir de esta variable, el programador puede realizar nuevas asignaciones para retener un estado particular del tablero y suplirla como argumento en funciones y procedimientos. Una asignación o la utilización de este valor como argumento significará una copia del objeto tablero.

5.4. Implementación del modo interactivo

Para la implementación del ciclo *read-eval-print* del modo interactivo utilicé dos primitivas de entrada/salida, la función `read` y el procedimiento `Show`. Estas operaciones primitivas hacen uso de la interfaz abstracta `InteractiveAPI` para delegar sus implementaciones en instancias de esta interfaz que utilizan distintos medios de lectura de teclas y visualización de tableros.

5.4.1. Implementación del ciclo *read-eval-print*

El bloque `interactive program` se traduce internamente en un punto de entrada `program` mediante una etapa de expansión de macros que agregué al pipeline del compilador. Esta traducción genera un programa que implementa el comportamiento del ciclo *read-eval-print* y donde las asociaciones $(teclas, bloque)$ forman parte de la etapa *eval*. A continuación se presenta un esquema de esta traducción donde la anotación `KeyAssocs` denota una secuencia de asociaciones $(teclas, bloque)$.

```
# Esquema de programa
interactive program
 {
   <KeyAssocs>
 }

# Luego de la etapa de expansión de macros.
program
 {
   lastKey := read()

   while (lastKey /= K_CTRL_D)
    {
      switch (lastKey) to
        <KeyAssocs>
      Show()
      FreeVars()
      lastKey := read()
    }
 }
```

El programa expandido comienza asignando la variable `lastKey` con la lectura de la primer tecla obtenida a través de la primitiva `read`. A partir de este punto el programa ejecuta hasta que la variable `lastKey` denote el valor `K_CTRL_D`, combinación que establece el fin de transmisión. Esta ejecución consta de cuatro partes:

1. la alternativa indexada que ejecutará la secuencia de comandos correspondiente al valor de `lastKey` (etapa *eval*),
2. la llamada a la primitiva `Show` que mostrará el tablero (etapa *print*).
3. la llamada a la primitiva `FreeVars` que libera todas las variables presentes en el *scope*[6] actual, asegurando que el único medio de comunicación entre distintos ciclos de la ejecución sea el tablero.
4. La llamada a la primitiva `read` que efectuará una nueva lectura (etapa *read*).

[6]Contexto del programa. Incluye las variables y el tablero asociados a ese contexto.

A continuación se muestra un programa interactivo que permite manipular el cabezal en tiempo de ejecución mediante las flechas del teclado y su traducción mediante la etapa de expansión de macros.

```
# Programa original
interactive program
 {
   K_ARROW_LEFT  -> { Mover(Oeste) }
   K_ARROW_RIGHT -> { Mover(Este)  }
   K_ARROW_UP    -> { Mover(Norte) }
   K_ARROW_DOWN  -> { Mover(Sur)   }
   _ -> { Skip }
 }

# Luego de la etapa de expansión de macros.
program
 {
   lastKey := read()
   while (lastKey /= K_CTRL_D)
    {
      switch (lastKey) to
        K_ARROW_LEFT  -> { Mover(Oeste) }
        K_ARROW_RIGHT -> { Mover(Este)  }
        K_ARROW_UP    -> { Mover(Norte) }
        K_ARROW_DOWN  -> { Mover(Sur)   }
         _ -> { Skip }
      Show()
      FreeVars()
      lastKey := read()
    }
 }
```

Vemos claramente como la secuencia de asociaciones $(teclas, bloque)$ es colocada como cuerpo de la alternativa indexada.

5.4.2. La interfaz abstracta `InteractiveAPI`

La clase `InteractiveAPI` representa una interfaz para la abstracción de las etapas *read* (método `read`) y *print* (método `show`) del ciclo *read-eval-print* establecido por el modo interactivo, permitiendo desarrollar implementaciones para distintos medios de entrada y salida (ver figura 5.4.1). El método abstracto `read` supone la lectura de una tecla representada como un número y el método `show` recibe un tablero con el fin de ser mostrado. De esta manera, la máquina virtual delega la implementación de las primitivas `read` y `Show` en una instancia de esta interfaz, interacción que se ilustra en la figura 5.4.2.

Durante el desarrollo inicial de esta interfaz creé algunas implementaciones utilizadas para la realización de pruebas en la característica, que se describen a continuación:

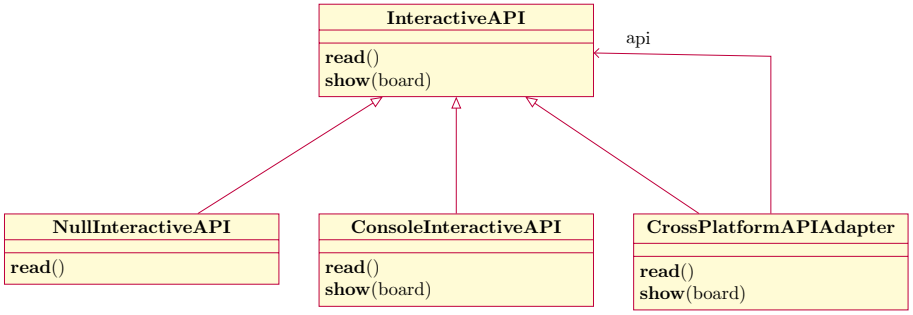


Figura 5.4.1: Diagrama de clase para la interfaz abstracta `InteractiveAPI` y sus implementaciones.

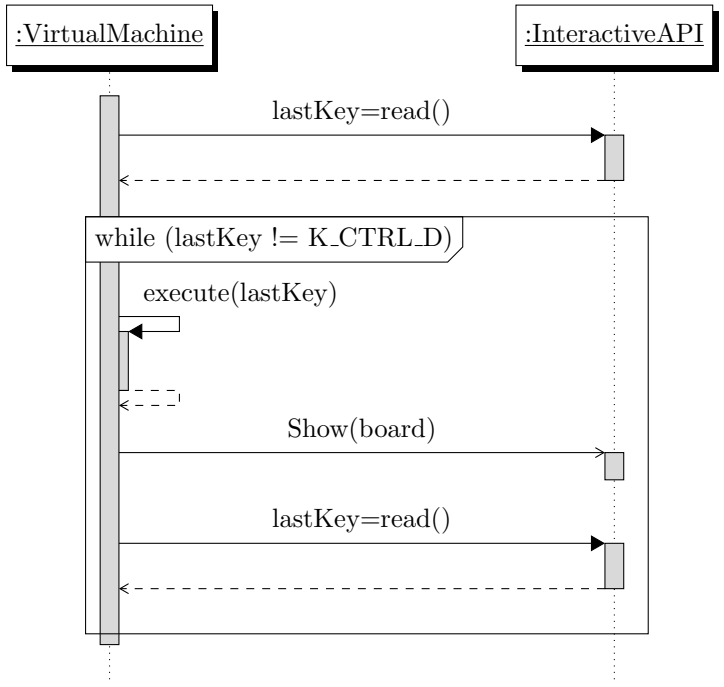


Figura 5.4.2: Interacción entre una máquina virtual y la interfaz interactiva

- `NullInteractiveAPI`. Alimenta la máquina virtual con una sucesión aleatoria de entradas, siendo `K_CTRL_D` la última entrada, y no implementa la primitiva *Show*.
- `ConsoleInteractiveAPI`. Toma caracteres ingresados por el usuario en la consola y utiliza el mismo medio para mostrar los resultados de cada ejecución parcial utilizando el formato visto en la sección 2.2.
- `CrossPlatformAPIAdapter`. Constituye un adaptador que, dado otra interfaz interactiva, permite detectar una secuencia de caracteres a fin de constituir teclas especiales como las flechas del teclado, teniendo en cuenta que las representaciones utilizadas por WINDOWS y LINUX.

En la sección siguiente explico de manera detalla la implementación de esta interfaz interactiva utilizada dentro del módulo de interacción entre las implementaciones de los lenguajes y la interfaz gráfica.

5.5. Módulo de interacción entre las implementaciones de los lenguajes y la interfaz gráfica

El módulo de interacción comprende una interfaz de abstracción que implementa, encapsula y oculta la concurrencia y el pasaje de mensajes entre el proceso que gestiona los compiladores y máquinas virtuales de GOBSTONES 3.0 y XGOBSTONES y el proceso que aloja la ejecución de la interfaz gráfica. Este módulo se estructura en tres partes:

- El proceso de la interfaz gráfica que hace uso de la interfaz del módulo.
- El proceso que envuelve la ejecución del compilador y máquina virtual de la implementación del lenguaje elegido.
- El mecanismo de comunicación entre los procesos.

En esta sección explicaré la implementación de cada una de estas.

5.5.1. Arquitectura del módulo

El diseño de la arquitectura del módulo está dado, a grandes rasgos, por tres componentes: la interfaz del módulo de la cual hace uso la interfaz gráfica de usuario, un gestor de las implementaciones que encapsula al compilador y la máquina virtual de GOBSTONES 3.0 y XGOBSTONES, y el mecanismo de comunicación entre estos.

La figura 5.5.1 presenta un diagrama de clases que muestra, de manera detallada, las clases y relaciones que componen el módulo de interacción como se describe a continuación.

La clase `GraphicalUserInterface` representa la interfaz gráfica de usuario.

La clase `ProgramRun` representa una ejecución particular de un programa. Posee el comportamiento necesario para la comunicación con el proceso que

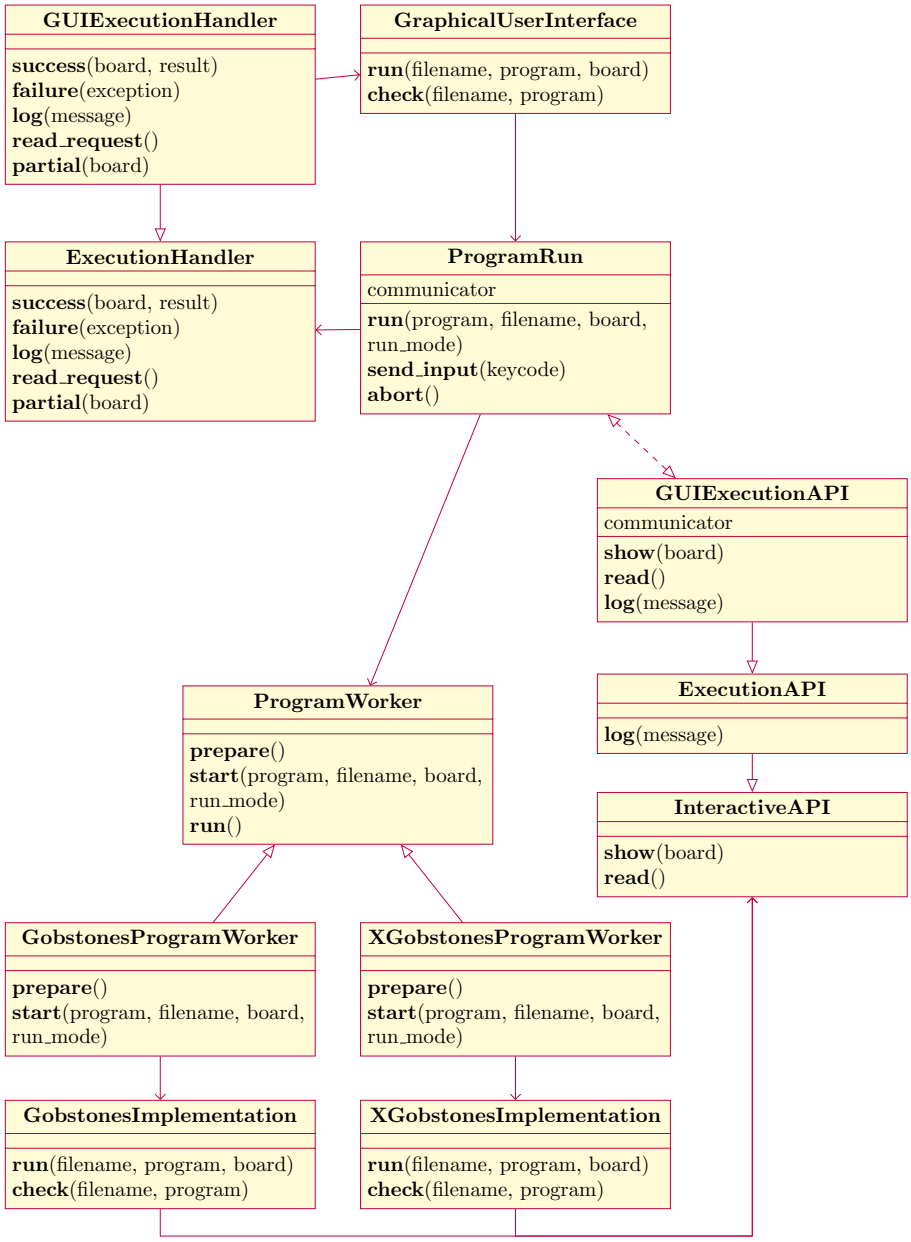


Figura 5.5.1: Diagrama de clases del módulo de interacción.

aloja al compilador y máquina virtual de la implementación del lenguaje elegido e implementa tres métodos que son de interés para el usuario de esta interfaz:

- `run` que dado un programa, el nombre del archivo que lo contiene, un tablero y un modo de ejecucion (ejecutar programa o verificar programa) comienza la ejecución del programa.
- `send_input` que dado un código de tecla (en codificación ASCII) envía a la máquina virtual una tecla a ser consumida por el modo interactivo.
- `abort` que detiene la ejecución.

La clase `ExecutionHandler` es utilizada por `ProgramRun` para delegar en ella el comportamiento relacionado con el fin de ejecución exitoso (`success`), el fallo en ejecución (`failure`), la atención de mensajes informativos (`log`), la lectura de una tecla (`read_request`) y la muestra de un resultado parcial (`partial`). Junto con la clase `ProgramRun` constituyen la interfaz del mecanismo de comunicación utilizada por la interfaz gráfica.

La clase `GUIExecutionHandler` es una implementación de la interfaz abstracta `ExecutionHandler` que interactúa con la interfaz gráfica para realizar las lecturas de teclado y mostrar los resultados de ejecución, resultados parciales, mensajes informativos y mensajes de fallo de ejecución.

La clase `ProgramWorker` es un gestor de las implementaciones del lenguaje que establece un ciclo de vida (mediante el método `run`) e implementa el comportamiento necesario para interactuar con la clase `ProgramRun` a través del mecanismo de comunicación. Esta clase es implementada por las clases `GobstonesProgramWorker` y `XGobstonesProgramWorker`, que contienen el comportamiento específico para la gestión de las implementaciones de los lenguajes GOBSTONES 3.0 y XGOBSTONES respectivamente.

Las clases `GobstonesImplementation` y `XGobstonesImplementation` constituyen un *facade* [4] de las implementaciones de los lengaujes de programación GOBSTONES 3.0 y XGOBSTONES respectivamente.

La clase `InteractiveAPI`, explicada en la sección 5.4, es subclasificada por `ExecutionAPI`, que agrega a la interfaz el método `log` para la comunicación de mensajes informativos por parte del compilador o la máquina virtual. A su vez, la clase `ExecutionAPI` es subclasficada por `GUIExecutionAPI` que implementa cada uno de los métodos a partir de mensajes que son remitidos mediante el mecanismo de comunicación del módulo.

5.5.2. Implementación de la interfaz del módulo

La interfaz del módulo de interacción está dada por la clase `ProgramRun`, que controla el mecanismo de comunicación e informa de los distintos eventos relacionados con la ejecución de las implementaciones del lenguaje, y la clase `ExecutionHandler`, que constituye una interfaz para la atención de los distintos eventos que anuncia `ProgramRun`.

La clase `ProgramRun` es responsable de la creación del proceso donde ejecutarán las implementaciones del lenguaje, de la emisión de un primer mensaje

que dé comienzo a la ejecución y del constante sondeo del canal de comunicación para detectar e informar a un manejador que implemente la interfaz `ExecutionHandler` de la presencia de nuevos eventos generados por la ejecución.

La interfaz establecida por la clase `ExecutionHandler` es implementada por la clase `GUIExecutionHandler` con el comportamiento específico para que la interfaz gráfica maneje cada uno de los eventos mencionados. De esta manera, la interfaz gráfica podrá mostrar un tablero parcial, pedir al usuario el ingreso de una tecla o informar del estado de la ejecución ignorando el mecanismo de comunicación subyacente.

5.5.3. Implementación del gestor de implementaciones

El gestor de implementaciones `ProgramWorker` es implementado por las clases `GobstonesProgramWorker` y `XGobstonesProgramWorker` que tienen la responsabilidad de instanciar el compilador y la máquina virtual de las respectivas implementaciones de los lenguajes y comunicar el resultado de la ejecución. Durante la instanciación proveen a las máquinas virtuales con una instancia de `GUIExecutionAPI` que implementa los métodos de la clase `InteractiveAPI` a través de envío y recepción de mensajes mediante el mecanismo de comunicación. Estas clases también son responsables de capturar las excepciones que puede levantar el compilador y la máquina virtual durante la ejecución de un programa. Estas excepciones se informan al proceso de la interfaz gráfica mediante un mensaje.

5.5.4. Comunicación entre la interfaz gráfica y las implementaciones de los lenguajes

En la figura 5.5.2 podemos ver un diagrama de secuencia que ilustra el mecanismo de comunicación utilizando en la interacción entre los componentes del módulo. La clase `ProgramRun` constituye la interfaz del mecanismo de comunicación empleada por la interfaz gráfica para interactuar con la implementación del lenguaje. La clase `Queue` es una cola de mensajes sincrónica implementada por PYTHON con primitivas de envío y recepción bloqueantes y no-bloqueantes. Esta cola constituye el canal de comunicación inter-proceso empleado. La clase `ProgramWorker` gestiona la implementación del lenguaje elegido. La interfaz gráfica junto con la instancia de `ProgramRun` ejecutan en un proceso separado de las implementaciones y sus gestores.

Una ejecución comienza a partir de la emisión de un mensaje de comienzo de ejecución (`START`) por parte de una instancia de `ProgramRun`. Este mensaje es recibido por una instancia de `ProgramWorker`, utilizando una primitiva de recepción bloqueante sobre el canal, que comienza la ejecución del compilador y la máquina virtual del lenguaje elegido, proveyendo a esta última de una instancia de `GUIExecutionAPI`. A partir de este punto la implementación del lenguaje dirige la comunicación mediante el envío sucesivo de mensajes a través de interfaz `GUIExecutionAPI`. Estos mensajes son recibidos por `ProgramRun` mediante

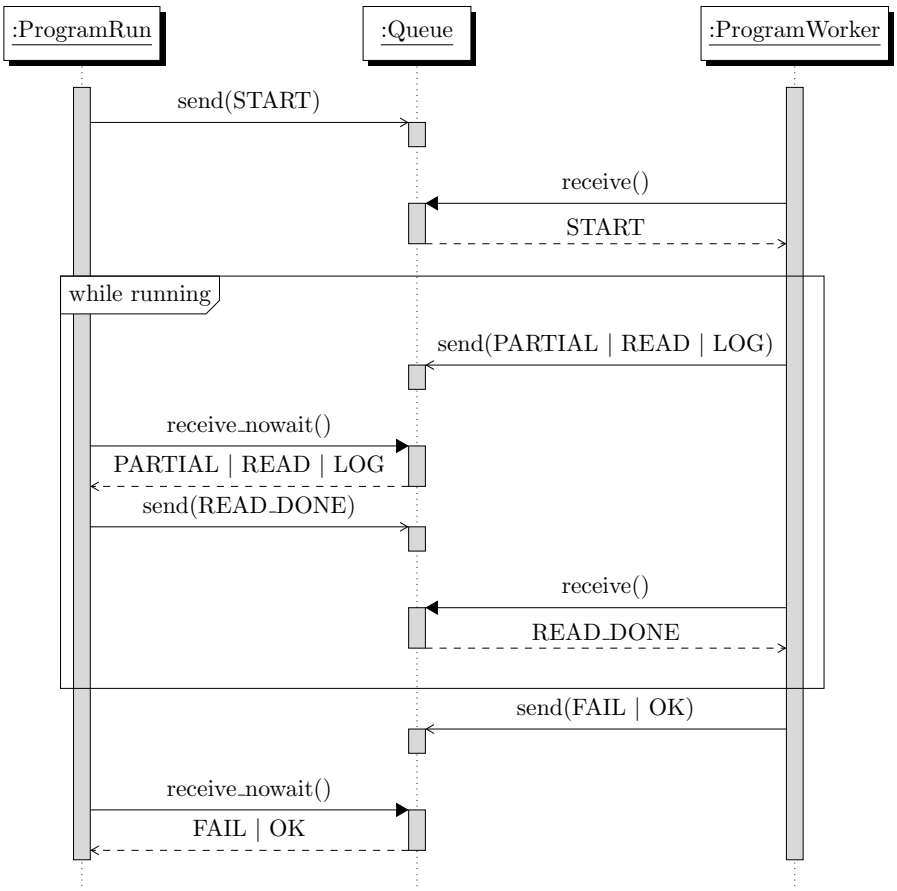


Figura 5.5.2: Interacción entre el proceso que aloja a la máquina virtual (gestionado por `ProgramWorker`) y el que aloja a la interfaz gráfica de usuario (cuya interfaz con el mecanismo de comunicación es `ProgramRun`).

polling [2] sobre el canal de comunicación, utilizando una primitiva de recepción no-bloqueante. De esta manera, la instancia de `ProgramRun` se mantiene a la espera de un requerimientos de lectura de tecla (`READ`), que responderá una vez realizada la lectura (`READ_DONE`), de muestra de resultado parcial (`PARTIAL`) o mensajes informativos respecto de la ejecución (`LOG`), hasta la llegada del mensaje de fin de ejecución exitosa (`OK`) o del mensaje de ejecución fallida (`FAIL`).

Los tipos de mensajes que se transmiten en este proceso de comunicación y los datos que incluyen son:

- `START`. Indica el comienzo de una ejecución. Incluye el nombre del archivo que contiene al programa, el programa, un tablero inicial[7] y el modo de ejecución.

- `FAIL`. Indica el fallo de la ejecución actual. Incluye la excepción que finalizó la ejecución.

- `OK`. Indica el fin de la ejecución actual de manera exitosa. Incluye el tablero final y una lista de tuplas $(variable, valor)$ que corresponden a las variables retornadas al finalizar la ejecución.

- `LOG`. Indica que el compilador o la máquina virtual produjeron un mensaje informativo. Incluye el mensaje.

- `READ`. Constituye un requerimiento de lectura de tecla por parte de la máquina virtual.

- `READ_DONE`. Responde al requerimiento de lectura de tecla de la máquina virtual. Incluye un código de tecla.

- `PARTIAL`. Indica que la máquina virtual produjo un tablero parcial. Incluye el tablero parcial.

5.5.5. Resumen

Una ejecución comienza mediante la invocación del método `run` de una instancia de `ProgramRun` por parte de la interfaz gráfica que envía un mensaje de comienzo a una instancia de `ProgramWorker` mediante el mecanismo de comunicación. Este mensaje dará comienzo a la compilación del programa incluido en éste y comenzará su ejecución en la máquina virtual de la implementación del lenguaje elegido. A partir de este punto, la máquina virtual envía sucesivos mensajes a la interfaz gráfica, a través de los componentes que ofician de interlocutores, requiriendo acciones o indicando el fin de la ejecución.

[7] En este proceso de comunicación los tableros se codifican en cadenas de caracteres de acuerdo al format GBB como se explica en el apéndice I.

6. Conclusiones

Este informe presentó el desarrollo de compiladores y máquinas virtuales que implementen los lenguajes de programación GOBSTONES 3.0 y XGOBSTONES y un mecanismo de comunicación entre estas implementaciones y la interfaz gráfica de la herramienta PYGOBSTONES 1.0 en el marco de las actividades del equipo de desarrollo de PYGOBSTONES.

Tras la realización de esta tesina he logrado desarrollar las implementaciones de los lenguajes de programación GOBSTONES 3.0 y XGOBSTONES que han sido propuestas junto a un mecanismo de interacción apropiado para la comunicación de estas implementaciones con la interfaz gráfica creada por el equipo de desarrollo de PYGOBSTONES. Estos desarrollos completan la primer versión de PYGOBSTONES 1.0, la herramienta producto de esta tesina. Actualmente está siendo utilizada en la Universidad Nacional de Quilmes para enseñar las nociones básicas de programación en la materia Introducción a la Programación. La herramienta PYGOBSTONES 1.0 es de código abierto, se encuentra publicada bajo la licencia GNU GPLv3 y está disponible para su uso por cualquier persona interesada en la página oficial de GOBSTONES (`http://www.gobstones.org`).

El desarrollo de las implementaciones de los lenguajes implicó comprender las características de los procesos de análisis sintáctico y compilación y el funcionamiento de las máquinas virtuales para lograr la compilación de los lenguajes a bytecode y posteriormente ejecutarlos. Durante este proceso enfrenté diversos problemas de implementación, principalmente relacionados con la expresión de ideas abstractas, provistas por los lenguajes, mediante representaciones más simples.

Por otro lado, el desarrollo del módulo de interacción significó un gran desafío de diseño debido a que la interfaz gráfica debía permanecer independiente de la implementación del lenguaje, asegurando un alto rendimiento de la primera y una rápida respuesta ante los requerimientos de la segunda.

El marco de trabajo en el cual se desarrolló esta tesina implicó mantener una comunicación fluida con los miembros del equipo de desarrollo avocados a la implementación de la interfaz gráfica, estableciendo pautas para el diseño del mecanismo de comunicación, como así también compartiendo diversas especificaciones respecto de aquellos datos transversales tanto a las implementaciones de los lenguajes como a la interfaz gráfica.

Como trabajo a futuro se propone el desarrollo de implementaciones de las máquinas virtuales que apunten a alcanzar un rendimiento comparable con el rendimiento de la máquina virtual de PYTHON, el desarrollo de un proceso de compilación optimizante que reduzca los tiempos de ejecución de los programas y la unificación de las implementaciones de los lenguajes.

Referencias

[1] Alfred V. Aho, Jeffrey D. Ullman, Ravi Sethi y Monica S. Lam: *Compilers: Principles, Techniques, and Tools*. Addison Wesley, segunda edición, 2006, ISBN 978-0321486813.

[2] Andrews, Gregory R.: *Foundations of Multithreaded Parallel, and Distributed Programming*. Addison Wesley, 1999, ISBN 978-0201357523.

[3] Beck, Kent: *Test Driven Development: By Example*. Addison Wesley, primera edición, 2002, ISBN 978-0321146533.

[4] Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994, ISBN 978-0201633610.

[5] Ghezzi, Carlo y Mehdi Jazayeri: *Programming language concepts*. Wiley, tercera edición, 1997, ISBN 978-0471104261.

[6] Lutz, Mark: *Programming Python. Powerful Object-Oriented Programming*. O'Reilly Media, Diciembre 2010, ISBN 978-1449301835.

[7] Martínez López, Pablo E.: *Las bases conceptuales de la Programación. Una nueva forma de aprender a programar*. EBook. El autor, 2013, ISBN 978-9873340819. http://www.gobstones.org/?page_id=34.

[8] Thomas H. Cormen, Charles E. leiserson, Ronald L. Rivest y Clifford Stein: *Introduction to Algorithms*. The MIT Press, tercera edición, 2009, ISBN 978-0262033848.

A. Notación para la gramática BNF

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

$<conjunto>$

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

$<conjunto> \quad \rightarrow \quad definición$

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

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

y los símbolos en `negrita` determinan elementos finales.

Por ejemplo, la siguiente definición

$<conj> \quad \rightarrow \quad <elem_1>$ `->` $<elem_2>$
$<elem> \quad \rightarrow \quad$ `X` | `Y`

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

La notación permite además elementos opcionales, escrito como

$[<elemOpcional>]$

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

$(<conjBase>/(<conjAEliminar>))$

e incorpora notaciones para especificar cero o más apariciones de un elemento dado

$\{<elemento>\}^*$

y una o más apariciones de este elemento

$\{<elemento>\}+$

Por ejemplo, modificando la definición anterior a

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

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

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

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

B. Sintaxis de GOBSTONES 3.0

En esta sección se presenta la sintaxis de GOBSTONES en su versión 3.0.

B.1. Programas GOBSTONES 3.0

Un programa GOBSTONES 3.0 es un elemento del conjunto $<gobstones>$. Cada programa está conformado por un cuerpo principal (que puede ser plano o interactivo) y por una lista de definiciones de procedimientos y funciones.

$<gobstones>$ → $<programdef>$ $<defs>$ | $<defs>$ $<programdef>$

$<defs>$ → $<def>$ | $<def>$ $<defs>$
$<def>$ → `procedure` $<procName>$ $<params>$ $<procBody>$
| `function` $<funcName>$ $<params>$ $<funBody>$
$<programdef>$ → $<bprogdef>$ | $<iprogdef>$
$<bprogdef>$ → `program` $<programBody>$
$<iprogdef>$ → `interactive program` $<iprogBody>$

$<params>$ → $<varTuple>$

El cuerpo de un procedimiento es una lista de comandos encerrados entre llaves. El cuerpo de una función es similar, excepto que termina con el comando `return`. El cuerpo principal del programa en su versión plana tiene un `return` opcional, pero solo de variables. El cuerpo principal de un programa interactivo es diferente, y se compone exclusivamente de una asociación entre teclas y bloques de código.

$<procBody>$ → { $<cmds>$ }
$<funcBody>$ → { $<cmds>$ `return` $<gexpTuple1>$[;] }
$<programBody>$ → { $<cmds>$ [`return` $<gexpTuple1>$[;]] }
$<iprogBody>$ → { $<keyassocs>$ }

Los comandos se definen en la siguiente sección (C.3), las tuplas de expresiones y variables, en la sección C.6, y los nombres de funciones y procedimientos,

también en la sección C.6. Las asociaciones de teclas a bloques se definen en la sección B.4.

B.2. Comandos

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

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

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

```
<simplecmd>   →  Skip
              |  <procCall>
              |  <varName> := <gexp>
              |  <varTuple1> := <funcCall>
<procCall>    →  <proc> <args>
<proc>        →  <procName> | <predefProc>
<predefProc>  →  Poner | Sacar | Mover
              |  IrAlBorde | VaciarTablero
```

Las invocaciones a función y los argumentos para las invocaciones se describen en la sección C.5, y los nombres de variables en la sección C.6.

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

```
<compcmd>   →  if (<gexp>) [then] <blockcmd> [else <blockcmd>]
            |  switch(<gexp>) to <branches>
            |  repeat (<gexp>) <blockcmd>
            |  while (<gexp>) <blockcmd>
            |  foreach <varName> in <sequence> <blockcmd>
            |  <blockcmd>

<sequence>  →  [<seqdef>]
<seqdef>    →  <range> | <enum>
<enum>      →  <gexp> | <enum>,<gexp>
<range>     →  <gexp>..<gexp> | <gexp>,<gexp>..<gexp>
<branches>  →  _ -> <blockcmd>
            |  <lits> -> <blockcmd>[;] <branches>
<lits>      |  <literal> | <literal>, <lits>
```

El conjunto de literales *<literal>* se define en la sección C.5.

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

B.3. Expresiones

La expresiones se obtienen combinando ciertas formas básicas en distintos niveles. El nivel básico tiene las variables, las expresiones atómicas para indicar-

le al cabezal que cense del tablero, los literales, y las invocaciones de función y primitivas. Sobre ese nivel se construyen las expresiones aritméticas (sumas, productos, etc.) con la precedencia habitual. Sobre el nivel aritmético se construyen las expresiones relacionales (comparación entre números y otros literales) y sobre ellas, las expresiones booleanas (negación, conjunción y disyunción) también con la precedencia habitual.

```
<gexp>      →  <bexp>
<bexp>      →  <bterm> | <bterm>||<bexp>            (infixr)
<bterm>     →  <bfact> | <bfact>&&<bterm>           (infixr)
<bfact>     →  not <batom> | <batom>
<batom>     →  <nexp>
            |  <nexp><rop><nexp>
<nexp>      →  <nterm> | <nexp><nop><nterm>         (infixl)
<nterm>     →  <nfactH> | <nterm>*<nfactH>          (infixl)
<nfactH>    →  <nfactL> | <nfactL><mop><nfactL>
<nfactL>    →  <natom> | <nfactL>^<natom>           (infixl)

<natom>     →  <varName> | <liter> | -<natom>
            |  <funcCall>
            |  (<gexp>)

<rop>       →  == | /= | < | <= | >= | >
<nop>       →  + | -
<mop>       →  div | mod

<funcCall>  →  <func> <args>
<args>      →  <gexpTuple>
<func>      →  <funcName> | <predefFunc>
<predefFunc> →  nroBolitas | hayBolitas | puedeMover
             |  siguiente | previo | opuesto
             |  minBool | maxBool
             |  minDir | maxDir
             |  minColor | maxColor
```

Las tuplas de expresiones se definen en la sección C.6.
Los literales pueden ser numéricos, booleanos, de color o de dirección.

```
<literal>   →  <literN> | <literB> | <literC> | <literD>
<literN>    →  <num>
<literB>    →  False | True
<literC>    →  Verde | Rojo | Azul | Negro
<literD>    →  Norte | Sur | Este | Oeste
```

La forma de los números se definen en la sección C.7

B.4. Programas interactivos

Los programas interactivos quedan definidos por una asociación entre especificaciones de teclas y bloques de código. La definición es la siguiente:

```
<keyassocs>         →  [<defaultkeyassoc>] | <keyassoc><keyassocs>
<keyassoc>          →  <keydef> -> <blockcmd>
<defaultkeyassoc>   →  _ -> <blockcmd>
```

Los bloques se definieron en la sección C.3. Las especificaciones de teclas se definen en la sección C.7.

B.5. Definiciones auxiliares

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

```
<varName>    →  <lowerid>
<funcName>   →  <lowerid>
<procName>   →  <upperid>
```

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

```
<varTuple>    →  () | <varTuple1>
<varTuple1>   →  (<varNames>)
<varNames>    →  <varName> | <varName>,<varNames>

<gexpTuple>   →  () | <gexpTuple1>
<gexpTuple1>  →  (<gexps>)
<gexps>       →  <gexp> | <gexp>,<gexps>
```

B.6. Definiciones lexicográficas

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

Los números son simplemente secuencias de dígitos.

```
<num>        →  <digit> | <nonzerodigit> <digits> | -<num>
<digits>     →  <digit> | <digit> <num>
<digit>      →  0 | <nonzerod>
<nonzerod>   →  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

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

```
<lowerid>     →  (<lowname>/(<reservedid>))
<lowname>     →  <lowchar> | <lowchar> <chars>
<lowchar>     →  a | ... | z

<upperid>     →  (<uppname>/(<reservedid>))
<uppname>     →  <uppchar> | <uppchar> <chars>
<uppchar>     →  A | ... | Z

<chars>       →  <char> | <char> <chars>
<basicchar>   →  <lowchar> | <uppchar> | <digit>
<char>        →  <basicchar> | ' | _
```

Las especificaciones de teclas son palabras reservadas que empiezan con `K_` (por tecla en inglés, *key*), y pueden ser las siguientes

```
<keydef>      →  K_<key>
<key>         →  <uppchar> | <digit> | <specialkey> | <arrowkey>
<specialkey>  →  SPACE | ENTER | TAB | BACKSPACE | DELETE | ESCAPE | CTRL_D
<arrowkey>    →  ARROW_LEFT | ARROW_RIGHT | ARROW_UP | ARROW_DOWN
```

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

```
<reservedid>  →  if | then | else | not | True | False
              |  Verde | Rojo | Azul | Negro
              |  Norte | Sur | Este | Oeste
              |  switch | to | while | repeat | Skip | foreach | in
              |  interactive | program | procedure | function | return
              |  Mover | Poner | Sacar | IrAlBorde | VaciarTablero
              |  div | mod | siguiente | previo | opuesto
              |  hayBolitas | nroBolitas | puedeMover
              |  minBool | maxBool
              |  minDir | maxDir
              |  minColor | maxColor
<reservedop>  →  := | .. | _ | ->
              |  , | ; | ( | ) | { | } | [ | ]
              |  || | && | + | * | - | ^
              |  == | /= | < | <= | >= | >
              |  -- | {- | -}
              |  // | /* | */
              |  #  | """
```

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

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

C. Sintaxis de XGOBSTONES

En esta sección se presenta la sintaxis de XGOBSTONES en su versión 1.0.

C.1. Programas XGOBSTONES

Un programa XGOBSTONES es un elemento del conjunto *<xgobstones>*. Cada programa está conformado por un cuerpo principal y por una lista de definiciones, que incluye procedimientos, funciones y tipos.

<xgobstones> → *<defs>*

<defs> → *<def>* | *<def><defs>*
<def> → `procedure` [*<varName>*.]*<procName>* *<params>* *<procBody>*
| `function` *<funcName>* *<params>* *<funBody>*
| `type` *<typeName>* `is` *<typedef>*
| [*<varName>*.]`program` *<programBody>*
| `interactive` [*<varName>*.]`program` *<iprogBody>*

Los procedimientos llevan siempre un primer parámetro especial consistente en una variable.

<params> → *<varTuple>*

El cuerpo de un procedimiento es una lista de comandos encerrados entre llaves.

<procBody> → { *<cmds>* }

El cuerpo de una función es similar, excepto que termina con el comando `return`.

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

El cuerpo principal del programa tiene un `return` opcional, pero solo de variables.

<programBody> → { *<cmds>* [`return` *<gexpTuple1>*[;]] }

Los comandos se definen en la sección C.3, las tuplas de expresiones y variables, en la sección C.6, y los nombres de funciones y procedimientos, también en la sección C.6. Las declaraciones de tipos se definen en la siguiente sección (C.2).

C.2. Declaraciones de tipos

Las declaraciones de tipos en XGOBSTONES son de dos formas posibles: registros y variantes.

⟨*typedef*⟩ → ⟨*recordtype*⟩ | ⟨*varianttype*⟩

Los tipos registro son estructuras que agrupan varios componentes en *campos* (en inglés, *fields*), identificados mediante una forma especial de nombres, los *nombres de campos* (a veces denominados *tags*). La declaración de un tipo registro indica los campos con sus tags, y opcionalmente indica un tipo para ellos.

⟨*recordtype*⟩ → `record` ⟨*recordDef*⟩
⟨*recordDef*⟩ → {⟨*fieldDefs*⟩}
⟨*fieldDefs*⟩ → ⟨*fieldDef*⟩ | ⟨*fieldDef*⟩[`,`]⟨*fieldDefs*⟩
⟨*fieldDef*⟩ → `field` ⟨*fieldName*⟩

Los tipos variantes son estructuras que aceptan diversos casos (*cases*), cada uno de los cuales se identifica mediante un nombre especial, el *constructor*, el cual puede, opcionalmente, poseer algunos campos.

⟨*varianttype*⟩ → `variant` ⟨*variantDef*⟩
⟨*variantDef*⟩ → {⟨*caseDefs*⟩}
⟨*caseDefs*⟩ → ⟨*caseDef*⟩ | ⟨*caseDef*⟩[`|`]⟨*caseDefs*⟩
⟨*caseDef*⟩ → `case` ⟨*caseName*⟩ [{⟨*fieldDefs*⟩}]

Los nombres de tipos, de campos y de casos se definirán en la sección C.6.

C.3. Comandos

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

⟨*blockcmd*⟩ → { ⟨*cmds*⟩ }
⟨*cmds*⟩ → [⟨*necmds*⟩[`;`]]
⟨*necmds*⟩ → ⟨*cmd*⟩ | ⟨*cmd*⟩[`;`] ⟨*necmds*⟩
⟨*cmd*⟩ → ⟨*simplecmd*⟩ | ⟨*compcmd*⟩

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

⟨*simplecmd*⟩ → `Skip`
| ⟨*procCall*⟩
| ⟨*variable*⟩ `:=` ⟨*gexp*⟩ | ⟨*varTuple1*⟩ `:=` ⟨*funcCall*⟩
⟨*procCall*⟩ → [⟨*variable*⟩`.`]⟨*proc*⟩⟨*args*⟩
⟨*proc*⟩ → ⟨*procName*⟩ | ⟨*predefProc*⟩
⟨*variable*⟩ → ⟨*varName*⟩ | ⟨*variable*⟩`[`⟨*gexp*⟩`]`
| ⟨*variable*⟩`.`⟨*predefField*⟩ | ⟨*variable*⟩`.`⟨*fieldName*⟩

Los procedimientos, que siempre tienen el primer parámetro por referencia, se invocan con ese primer parámetro del lado izquierdo, y los restantes como argumentos tradicionales. De esta manera, los efectos del procedimiento afectan solamente al parámetro suministrado como referencia. Los procedimientos predefinidos (⟨*predefProc*⟩) se especifican en la sección C.4.

Las invocaciones a función y los argumentos para las invocaciones se describen en la sección C.5, y los nombres de variables en la sección C.6.

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

```
<compcmd>  →  if (<gexp>) [then] <blockcmd> [else <blockcmd>]
           |  switch(<gexp>) to <branches>
           |  repeat (<gexp>) <blockcmd>
           |  while (<gexp>) <blockcmd>
           |  foreach <varName> in <gexp> <blockcmd>
           |  <blockcmd>

<branches> →  _ -> <blockcmd>
           |  <lits> -><blockcmd>[;] <branches>
<lits>     |  <literal> | <literal>, <lits>
```

El conjunto de literales <*literal*> se define en la sección C.5.

Observar que las condiciones de las alternativas y de la repetición condicional deben ir obligatoriamente entre paréntesis. También observar que el `foreach` acepta expresiones para especificar la secuencia de operaciones; esta expresión deberá proveer una lista.

C.4. Procedimientos predefinidos

Existen una serie de procedimientos predefinidos para el tipo de los tableros.

```
<predefProc>  →  Poner | Sacar | Mover
              |  IrAlBorde | VaciarTablero
```

En todos los casos se utilizan con la sintaxis de un llamado a procedimiento, usando como parámetro por referencia a una variable del tipo correspondiente (que no se consigna en esta sección, sino en C.3, con las llamadas a procedimientos). Las expresiones se definen en la sección siguiente.

C.5. Expresiones

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

```
<gexp>     →  <bexp> | <matchexp>
<bexp>     →  <bterm> | <bterm>||<bexp>          (infixr)
<bterm>    →  <bfact> | <bfact>&&<bterm>         (infixr)
<bfact>    →  not <batom> | <batom>
<batom>    →  <lexp>
           |  <lexp><rop><lexp>
<lexp>     →  <nexp> | <lterm>++<lexp>           (infixr)
<lterm>    →  <nexp>
<nexp>     →  <nterm> | <nexp><nop><nterm>       (infixl)
<nterm>    →  <nfactH> | <nterm>*<nfactH>        (infixl)
<nfactH>   →  <nfactL> | <nfactL><mop><nfactL>
<nfactL>   →  <natom> | <nfactL>^<natom>         (infixl)
```

Las expresiones de alternativa indexada se definen como sigue.

```
<matchexp>     →  match(<gexp>) to <mbranches>
<mbranches>    →  <mbranch> | <defaultmb>
               |  <mbranch><mbranches>
<defaultmb>    →  _ -> <gexp>
<mbranch>      →  <caseName> -> <gexp>
```

Las expresiones atómicas quedan definidas de la siguiente manera.

```
<natom>  →  <varName> | <liter> | -<natom>
         |  <funcCall>
         |  <listAtom>
         |  <constrAtom>
         |  <gexp>.<fieldName>
         |  <gexp>[<fieldName>]
         |  (<gexp>)
```

Los operadores y los llamados a función se definen como sigue.

```
<rop>          →  == | /= | < | <= | >= | >
<nop>          →  + | -
<mop>          →  div | mod

<funcCall>     →  <funcName> <gexpTuple> | <fieldName>(<gexp>)
               |  <predefFunc0>()
               |  <predefFunc1>(<gexp>)
               |  <predefFunc2>(<gexp>,<gexp>)

<predefFunc0>  →  minBool | maxBool
               |  minDir | maxDir
               |  minColor | maxColor
<predefFunc1>  →  head | tail | last | init | size
               |  siguiente | previo | opuesto
<predefFunc2>  →  nroBolitas | hayBolitas | puedeMover
```

Las tuplas de expresiones se definen en la sección C.6.

Los átomos de listas se definen ya sea mediante un rango (que puede tener un paso opcional), o mediante una enumeración. Los átomos de registros, variantes, arreglos y el tablero se definen mediante el llamado al constructor correspondiente.

```
<listAtom>      →  [<seqdef>]
<seqdef>        →  <range> | <enum>
<enum>          →  <gexp> | <enum>,<gexp>
<range>         →  <gexp>..<gexp> | <gexp>,<gexp>..<gexp>

<constrAtom>    →  Arreglo(size <- <gexp>)
                |  <constrName>(<fieldAssocs>)
<constrName>    →  <typeName> | <caseName>
<fieldAssocs>   →  <fieldAssoc> | <fieldAssoc><fieldAssocs>
<fieldAssoc>    →  <fieldName> <- <gexp>
```

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

```
<literal>  →  <literS> | <literN> | <literB> | <literC> | <literD>
<literS>   →  <string>
<literN>   →  <num>
<literB>   →  False | True
<literC>   →  Verde | Rojo | Azul | Negro
<literD>   →  Norte | Sur | Este | Oeste
```

La forma de los números y los strings se define en la sección C.7

C.6. Definiciones auxiliares

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

```
<varName>    →  <lowerid>
<funcName>   →  <lowerid>
<procName>   →  <upperid>
<typeName>   →  <upperid>
<fieldName>  →  <lowerid>
<caseName>   →  <upperid>
```

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

```
<varTuple>    →  () | <varTuple1>
<varTuple1>   →  (<variables>)
<variables>   →  <variable> | <variable>,<variables>

<gexpTuple>   →  () | <gexpTuple1>
<gexpTuple1>  →  (<gexps>)
<gexps>       →  <gexp> | <gexp>,<gexps>
```

C.7. Definiciones lexicográficas

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

Los números son simplemente secuencias de dígitos.

```
<num>       →  <digit> | <nonzerodigit> <digits> | -<num>
<digits>    →  <digit> | <digit> <num>
<digit>     →  0 | <nonzerod>
<nonzerod>  →  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

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

```
<lowerid>   →  (<lowname>/(<reservedid>))
<lowname>   →  <lowchar> | <lowchar> <chars>
<lowchar>   →  a | b | c | d | e | f | g | h | i | j | k | l | m
            |  n | o | p | q | r | s | t | u | v | w | x | y | z

<upperid>   →  (<uppname>/(<reservedid>))
<uppname>   →  <uppchar> | <uppchar> <chars>
<uppchar>   →  A | B | C | D | E | F | G | H | I | J | K | L | M
            |  N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<chars>     →  <char> | <char> <chars>
<basicchar> →  <lowchar> | <uppchar> | <digit>
<char>      →  <basicchar> | ' | _
```

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

```
<string>  →  " (<anySymbols>/(")) "
```

Algunos caracteres pueden usarse precedidos por una barra (\) llamada escape. Dentro de un string, la barra \\ y las comillas dobles \" siempre deben escaparse. La categoría de caracteres escapados incluye representaciones portables para los caracteres "alert"(\a), "backspace"(\b), "form feed" (\f), "new line" (\n), "carriage return"(\r), "horizontal tab" (\t), and "vertical tab" (\v). Observar que las comillas escapadas no son consideradas comillas, por lo que pueden usarse dentro de un string.

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

```
<reservedid>  →  if | then | else | not | True | False
              |  Verde | Rojo | Azul | Negro
              |  Norte | Sur | Este | Oeste
              |  Skip | switch | match | to | while | foreach | in | repeat
              |  program | type | is | procedure | function | return
              |  Mover | Poner | Sacar | VaciarTablero | IrAlBorde
              |  record | field | variant | case
              |  div | mod | siguiente | previo | opuesto
              |  head | tail | last | init | size
              |  hayBolitas | nroBolitas | puedeMover
              |  minBool | maxBool | minDir | maxDir
              |  minColor | maxColor
              |  Arreglo
```

```
<reservedop>  →  := | .. | ++ | . | _ | -> | <-
              |  , | ; | | | ( | ) | { | } | [ | ] | "
              |  || | && | + | * | - | ^
              |  == | /= | < | <= | >= | >
              |  -- | {- | -} | // | /* | */ | #  | """
```

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

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

D. El bytecode de GOBSTONES 2.0

Una instrucción de este bytecode se define por la siguiente gramática BNF (las características de esta notación se explican en el apéndice A):

```
<instruction_name>      →  pushConst | pushVar | delVar | assign
                        |  call | BOOM | label | enter | leave
                        |  jump | jumpIfNotIn | jumpIfFalse
                        |  return | returnVars | procedure
                        |  function | end

<bytecode_instruction>  →  <instruction_name> {<parameter>}*

<parameter>             →  ({<alphanum>}+/(<instruction_name>))

<alphanum>              →  <digit> | <char>
<digit>                 →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<char>                  →  <lowchar> | <uppchar>
<lowchar>               →  a | b | c | d | e | f | g | h | i | j | k | l | m
                        |  n | o | p | q | r | s | t | u | v | w | x | y | z
<uppchar>               →  A | B | C | D | E | F | G | H | I | J | K | L | M
                        |  N | O | P | Q | R | S | T | U | V | W | X | Y | Z
```

El comportamiento de cada una de estas instrucciones se describe a continuación.

`pushConst`

Agrega una constante a la pila del programa.

`pushVar`

Agrega el valor que denota un identificador a la pila del programa.

`delVar`

Elimina un identificador.

`assign`

Asigna una expresión a un identificador.

`call`

Invoca a una rutina del programa.

`BOOM`

Desencadena un error que termina la ejecución.

`label`

Es una etiqueta que puede ser referenciada por las instrucciones de salto.

`jump`

Posiciona el *ip* del programa en una etiqueta dada.

`jumpIfFalse`

Posiciona el *ip* del programa en una etiqueta dada si el último elemento de la pila del programa es False.

`jumpIfNotIn`

Posiciona el *ip* del programa en una etiqueta dada si el último elemento de la pila del programa pertenece a un conjunto de literales determinado.

`enter`

Inicializa un nuevo estado global guardando el anterior.

`leave`

Retorna al estado global anterior.

`return`

Descarta el registro de activación actual reemplazándolo por el que se encuentra en el tope de la pila de ejecución.

`returnVars`

Indica el fin de la ejecución y los identificadores que deben formar parte del resultado.

`procedure`

Indica el comienzo de una rutina de tipo “procedure”.

`function`

Indica el comienzo de una rutina de tipo “function”.

`end`

Indica el fin del código de una rutina.

E. El bytecode de GOBSTONES 3.0 y XGOBSTONES

El bytecode de GOBSTONES 3.0 es una versión modificada del bytecode de GOBSTONES 2.0 introducido en el apéndice D.

<*instruction_name*>	→	`pushConst` | `pushFrom` | `popTo` | `delVar`
	|	`THROW_ERROR` | `call` | `label` | `enter` | `leave`
	|	`jump` | `jumpIfNotIn` | `jumpIfFalse`
	|	`return` | `returnVars` | `procedure`
	|	`function` | `entrypoint` | `end`
	|	`setImmutable` | `unsetImmutable`
<*bytecode _instruction*>	→	<*instruction_name*> {<*parameter*>}*
<*parameter*>	→	({<*alphanum*>}+/(<*instruction_name*>))
<*alphanum*>	→	<*digit*> | <*char*>
<*digit*>	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
<*char*>	→	<*lowchar*> | <*uppchar*>
<*lowchar*>	→	`a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j` | `k` | `l` | `m`
	|	`n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` | `v` | `w` | `x` | `y` | `z`
<*uppchar*>	→	`A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M`
	|	`N` | `O` | `P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z`

A continuación se describe el comportamiento de las nuevas instrucciones.

`pushFrom`

Renombre de la instrucción `pushVar`.

`popTo`

Renombre de la instrucción `assign`.

`THROW_ERROR`

Renombre de la instrucción `BOOM`

`entrypoint`

Indica el comienzo de un punto de entrada.

`setImmutable`

Dado un nombre de variable, indica a la máquina virtual que ese nombre no puede ser modificado.

`unsetImmutable`

Dado un nombre de variable, indica a la máquina virtual que ese nombre puede ser modificado.

El resto de las instrucciones se comportan de manera análoga a la de su contraparte en GOBSTONES 2.0.

F. Gramática BNF utilizada por PyGobstones 0.97

La herramienta PyGobstones 0.97 utiliza una gramática BNF, contenida en un archivo de texto plano, con la cual realiza el parseo de un programa Gobstones 2.0. La sintaxis de este archivo se describe a continuación.

```
<rule>          →   <ruleName> ::= <expressions> ;;
<ruleName>      →   {<ruleChar>}+
<expressions>   →   <expression>
                |   <expression> | <expressions>
<expression>    →   <term>
                |   <ruleName>
                |   <expression> <expression>
<term>          →   ({<termChar>}+/(<reservedId>))

<ruleChar>      →   <alphanum> | < | > | _
<termChar>      →   <specialChar> | <alphanum>
<specialChar>   →   _ | - | < | > | = | : | & | + | - | * | /
                |   ; | . | , | { } | ( | ) | [ | ] | | | "
                |   ~ | ' | ? | ¿ | ¡ | ^ | @ | $ | ! | %
<alphanum>      →   <digit> | <char>
<digit>         →   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<char>          →   <lowchar> | <uppchar>
<lowchar>       →   a | b | c | d | e | f | g | h | i | j | k | l | m
                |   n | o | p | q | r | s | t | u | v | w | x | y | z
<uppchar>       →   A | B | C | D | E | F | G | H | I | J | K | L | M
                |   N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<reservedId>    →   ::= | | | ;; | #
```

Adicionalmente es posible especificar comentarios de línea utilizando el carácter `#`.

```
<comment>   →   #(<anySymbols>/(\n)) \n
```

El siguiente ejemplo muestra una gramática BNF, según la notación que emplea Gobstones 2.0, que reconoce una forma básica de lambda calculus.

```
<expression> ::= <variable>
               | ( <expression> <expression> )
               | ( \ <variable> . <expression> )
             ;;

<variable>   ::= [a-z]+
```

Esta sintaxis se utiliza en el apéndice G.1 para mostrar los cambios que se hicieron a las gramáticas BNF utilizadas por los compiladores de Gobstones 3.0 y XGobstones para realizar el análisis sintáctico de programas.

G. Implementación de GOBSTONES 3.0

Esta sección incluye parte de la implementación de GOBSTONES 3.0.

G.1. Cambios de sintaxis

Mostramos la sintaxis de los cambios introducidos en GOBSTONES 3.0. Se incluirán comparaciones con la sintaxis de GOBSTONES 2.0.

G.1.1. Punto de entrada `program`

Se muestra la incorporación del punto de entrada `program` al conjunto de definiciones del lenguaje.

```
# Gobstones 2.0
<def> ::= function <funcName> <params> <procFuncBody>
        | procedure <procName> <params> <procFuncBody>
      ;;
<procFuncBody> ::= { <cmds> <return?> } ;;


## Gobstones 3.0
<def> ::= function <funcName> <params> <procFuncBody>
        | procedure <procName> <params> <procFuncBody>
        | program <procFuncBody>
      ;;
<procFuncBody> ::= { <cmds> <return?> } ;;
```

G.1.2. Estructura `if-then-else`

Se muestra el cambio de sintaxis en la alternativa condicional.

```
# Gobstones 2.0
<ifCmd> ::= if ( <gexp> ) <blockcmd> <else?> ;;
<else?> ::= <EMPTY>
          | else <blockcmd>
        ;;

# Gobstones 3.0
<ifCmd> ::= if ( <gexp> ) <then?> <blockcmd> <else?> ;;
<then?> ::= <EMPTY>
          | then
        ;;
<else?> ::= <EMPTY>
          | else <blockcmd>
        ;;
```

G.1.3. Estructura `switch-to`

Se muestra el cambio de sintaxis relacionado con la alternativa indexada.

```
# Gobstones 2.0
<caseOfCmd>   ::= case ( <gexp> ) of <branches> ;;

# Gobstones 3.0
<switchToCmd> ::= switch ( <gexp> ) to <branches> ;;
```

G.1.4. Comentarios de línea y multi-línea

Se muestra la sintaxis que define los comentarios en GOBSTONES 3.0 y se compara con la forma en que los define GOBSTONES 2.0.

```
# Gobstones 2.0
COMMENT ::= --[^\n]*
        | //[^\n]*
        | {-([^-]|-+[^}-])*-+}
        | /[*]([^*]|[*]+[^*/])*[*]+/
      ;;

## Gobstones 3.0
COMMENT ::= --[^\n]*
        | //[^\n]*
        | #[^\n]*
        | {-([^-]|-+[^}-])*-+}
        | /[*]([^*]|[*]+[^*/])*[*]+/
        | """([^"]|"[^"]|""[^"])*"""
      ;;
```

G.1.5. Repetición simple

Se muestra la sintaxis de la repetición simple de GOBSTONES 3.0.

```
## Gobstones 3.0
<repeatCmd> ::= repeat ( <gexp> ) <blockcmd> ;;
```

G.1.6. Repetición indexada, rangos y secuencias explícitas

Se muestra la sintaxis de la repetición indexada en conjunto con los rangos simples y secuencias explícitas. A su vez se compara la sintaxis GOBSTONES 3.0 con la de GOBSTONES 2.0.

```
# Gobstones 2.0
<repeatWithCmd> ::= repeatWith <varName> in <range>
                    <blockcmd> ;;
<range> ::= <gexp> .. <gexp> ;;
```

```
## Gobstones 3.0
<foreachCmd>   ::= foreach <varName> in <rangeOrSeq>
                     <blockcmd> ;;
<rangeOrSeq>  ::= [ <gexp> <,gexp?> <rangeOrSeq1> ] ;;
<rangeOrSeq1> ::= , <gexps+>
                 | .. <gexp>
                ;;
<,gexp?> ::= <EMPTY>
           | , <gexp>
         ;;
```

H. Programas GOBSTONES

Esta sección se presentan programas GOBSTONES que complementan los ejemplos presentados.

H.1. Ejemplo de un programa en GOBSTONES 2.0

El siguiente programa tiene por objetivo dibujar un cuadrado rojo de lado tres y un cuadrado verde de lado dos utilizando bolitas.

```
procedure MoverN(n, d)
 {
   repeatWith i in 1..n
    { Mover(d) }
 }

procedure DibujarLineaNDeColorAl(n, c, d)
 {
   repeatWith i in 1..n-1
    { Poner(c); Mover(d) }
   Poner(c)
 }

procedure DibujarCuadradoDeLadoNYColor(n, c)
 {
   repeatWith d in minDir()..maxDir()
    { DibujarLineaNDeColorAl(n, c, d) }
 }

function esCeldaVacia()
 {
   esVacia := True
   repeatWith c in minColor()..maxColor()
    { esVacia := esVacia && not hayBolitas(c) }
   return(esVacia)
 }
```

```
function celdasOcupadasAl(d)
 {
   ocupadas := 0
   while (not esCeldaVacia())
    {
      ocupadas := ocupadas + 1
      Mover(d)
    }
   return(ocupadas)
 }

procedure NuevoAreaDeDibujoAl(d)
 { MoverN(celdasOcupadasAl(d), d) }

procedure Main()
 {
   VaciarTablero()
   IrAlOrigen()
   DibujarCuadradoDeLadoNYColor(3, Rojo)
   NuevoAreaDeDibujoAl(Este)
   DibujarCuadradoDeLadoNYColor(2, Verde)
 }
```

H.2. Pequeño juego en GOBSTONES 3.0

El pequeño juego que presento a continuación consta de un personaje que se va desplazando por el tablero con el objetivo de recolectar todas las monedas mientras evita las bombas. El juego se gana recolectando todas las monedas. Si el personaje pisa una bomba, el jugador pierde y el juego finaliza.

```
/* Gobstones Little Game.
 ************************************************
 * Juego básico en Gobstones en el cuál se muestra
 * como manipular un personaje utilizando el modo
 * interactivo. El objetivo del juego es agarrar
 * todas las monedas evitando pisar las bombas.
 *
 * Autor: Ary Pablo Batista <arypbatista@gmail.com>
 */

interactive program
 /*
   PROPóSITO: Inicializa el juego al presionar la
   tecla ENTER. Mueve al personaje utilizando las
   flechas del teclado.
 */
```

```
 {
   # Movimiento del personaje
   K_ARROW_UP    -> { MoverPersonaje(Norte)
                      ChequearFinDelJuego() }
   K_ARROW_DOWN  -> { MoverPersonaje(Sur)
                      ChequearFinDelJuego() }
   K_ARROW_RIGHT -> { MoverPersonaje(Este)
                      ChequearFinDelJuego() }
   K_ARROW_LEFT  -> { MoverPersonaje(Oeste)
                      ChequearFinDelJuego() }

   # Control del juego
   K_ENTER -> { InicializarJuego() }
   _       -> { Skip }
 }

procedure InicializarJuego()
 /*
   PROPóSITO: Coloca monedas, bombas y al personaje
   en el tablero.
 */
 {
   VaciarTablero()

   # Pone bombas y monedas.
   contador := 1
   IrAPrimerCelda(Norte, Este)
   while (not esUltimaCelda(Norte, Este))
  {
     SiguienteCelda(Norte, Este)
     if (contador mod  7 == 0) { Poner(moneda()) }
     if (contador mod 13 == 0) { Poner(bomba())  }
     contador := contador + 1
   }

   # Coloca al personaje en el origen.
   IrAPrimerCelda(Norte, Este)
   Poner(personaje())
 }

function moneda()
 /* PROPóSITO: Denota a la entidad moneda */
 { return (Verde) }

function bomba()
 /* PROPóSITO: Denota a la entidad bomba */
 { return (Rojo) }
```

```
function personaje()
 /* PROPóSITO: Denota a la entidad personaje */
 { return (Azul) }

procedure MoverPersonaje(d)
 /*
   PROPóSITO: Mueve al personaje tomando monedas
   o destruyendo al personaje si se topa con una
   bomba.
 */
 {
   Sacar(personaje())
   Mover(d)
   if (esEntidad(moneda()))
  { Sacar(moneda()) }

   if (esEntidad(bomba()))
    { Sacar(bomba()) }
   else
    { Poner(personaje()) }
 }

function esEntidad(e)
 /*
   PROPóSITO: Indica si hay una entidad e en la
   celda actual.
 */
 { return (hayBolitas(e)) }

procedure ChequearFinDelJuego()
 /*
   PROPóSITO: Verifica si hay que finalizar el
   juego y muestra la pantalla de fin de juego.
 */
 {
   if (not personajeVivo())
  { Derrota() }
   else
  { if (not quedanMonedas()) { Victoria() } }
 }

function quedanMonedas()
 /*
   PROPóSITO: Indica si quedan monedas en
   el juego.
 */
 { return(hayBolitasEnTablero(moneda())) }
```

```
function personajeVivo()
 /*
   PROPóSITO: Indica si el personaje está
   vivo.
 */
 { return(hayBolitasEnTablero(personaje())) }


procedure Derrota()
 /*
   PROPóSITO: Muestra una pantalla indicando
   la derrota.
 */
 {
   VaciarTablero()
   LlenarTableroConBolitas(bomba())
 }


procedure Victoria()
 /*
   PROPóSITO: Muestra una pantalla indicando
   la victoria.
 */
 {
   VaciarTablero()
   LlenarTableroConBolitas(moneda())
 }


##########################
#   Biblioteca           #
##########################


procedure IrAPrimerCelda(dirInterna, dirExterna)
 /*
    PROPóSITO
      Denota la primer celda del tablero en un
      recorrido que recorre el tablero primero en
      dirección dirInterna y luego en dirección
      dirExterna.
 */
 {
   IrAlBorde(opuesto(dirInterna))
   IrAlBorde(opuesto(dirExterna))
 }
```

```
procedure LlenarTableroConBolitas(c)
 /*
   PROPóSITO
     Llena el tablero con bolitas
     de color c.
 */



 {
   IrAPrimerCelda(Norte, Este)
   Poner(c)
   while (not esUltimaCelda(Norte, Este))
    {
      SiguienteCelda(Norte, Este)
      Poner(c)
    }
 }



function esUltimaCelda(dirInterna, dirExterna)
 /*
    PROPóSITO
      Denota la última celda del tablero en un
      recorrido que recorre el tablero primero en
      dirección dirInterna y luego en dirección
      dirExterna.
 */
 { return (not puedeMover(Este) && not puedeMover(Norte)) }



procedure SiguienteCelda(dirInterna, dirExterna)
 /*
   PROPóSITO: Mueve el cabezal a la siguiente
   celda del tablero.
   PRECONDICIóN: Debe haber una celda en
   dirección dirInterna o dirExterna.
 */
 {
   if (puedeMover(dirInterna))
    { Mover(dirInterna) }
   else
    {
      IrAlBorde(opuesto(dirInterna))
      Mover(dirExterna)
    }
 }
```

```
function hayBolitasEnTablero(c)
 /*
   PROPóSITO: Indica si hay alguna bolita
   de color c en el tablero.
 */
 {
   IrAPrimerCelda(Norte, Este)
   hay := hayBolitas(c)
   while (not esUltimaCelda(Norte, Este))
   {
     SiguienteCelda(Norte, Este)
     hay := hay || hayBolitas(c)
   }
   return (hay)
 }
```

I. El formato GBB

Para representar los tableros mediante una cadena de caracteres se creó el formato GBB, actualmente en su versión 1.0, siendo utilizado para guardar tableros en archivos de texto plano o en la comunicación entre los módulos que componen PYGOBSTONES.

Un tablero en formato GBB v1.0 se define de la siguiente manera:

⟨*gbb-format*⟩ → ⟨*gbb-signature*⟩
⟨*gbb-size*⟩
{⟨*gbb-cell*⟩}*
⟨*gbb-head*⟩
⟨*gbb-closing*⟩

⟨*gbb-signature*⟩ → `GBB/1.0`
⟨*gbb-size*⟩ → ⟨*size-keyword*⟩ ⟨*x*⟩ ⟨*y*⟩
⟨*gbb-cell*⟩ → ⟨*cell-keyword*⟩ ⟨*x*⟩ ⟨*y*⟩ {⟨*color*⟩ ⟨*count*⟩}*
⟨*gbb-head*⟩ → ⟨*head-keyword*⟩ ⟨*x*⟩ ⟨*y*⟩
⟨*gbb-closing*⟩ → `%%`

⟨*color*⟩ → `Azul` | `Negro` | `Rojo` | `Verde`
⟨*cell-keyword*⟩ → `cell` | `o`
⟨*head-keyword*⟩ → `head` | `x`
⟨*size-keyword*⟩ → `size` | `s`
⟨*x*⟩ → ⟨*natural*⟩
⟨*y*⟩ → ⟨*natural*⟩

⟨*count*⟩ → ⟨*natural*⟩
⟨*natural*⟩ → {⟨*digit*⟩}+
⟨*digit*⟩ → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Un ejemplo de tablero escrito en este formato podría ser el siguiente:

```
GBB/1.0
size 9 9
cell 1 2 Rojo 23
cell 3 3 Negro 30 Verde 1
cell 8 8 Azul 2
head 1 2
%%
```