

Trabajo de Inserción Profesional

Título:

Desarrollo de un Entorno Integrado de Aprendizaje de Programación utilizando Editores Proyetivos siguiendo la didáctica de Gobstones

Alumno:

Ariel Alvarez

Director:

Ing. Nicolás Passerini

Codirector:

Ing. Javier Fernandes

Carrera:

Tecnicatura Universitaria en Programación Informática



Universidad
Nacional
de Quilmes

Índice

1. Introducción	1
1.1. Gobstones	2
1.2. Editores Proyectivos	2
1.3. Organización de este informe	3
2. Tecnología proyectiva a usar	4
3. Modelo conceptual del lenguaje Gobstones	4
4. Implementación del Editor	6
5. Implementación del Intérprete	8
5.1. Contexto de ejecución	9
5.2. Clases de dominio de Gobstones	10
5.3. Edición del tablero inicial	10
5.4. Renderización del tablero final	11
6. Chequeo de tipos	11
7. Mejorando la experiencia de Usuario	11
8. Definición de Ejercicios	14
9. Conocimientos Aplicados	15
10. Conclusión	15
11. Anexo	16
12. Agradecimientos	16

Desarrollo de un Entorno Integrado de Aprendizaje de Programación utilizando Editores Proyectivos siguiendo la didáctica de Gobstones

9 de febrero de 2016

Resumen

Gobstones constituye tanto un lenguaje de programación como secuencia didáctica bien definida que ha demostrado ser eficaz tanto en cursos iniciales universitarios como en escuelas secundarias [8]. En el marco de una comunidad creciente de usuarios [8], tanto por su uso en cursos universitarios como por la adopción de la secuencia didáctica por parte de los cursos de Program.Ar, se desarrolló una primera versión de un *entorno integrado de aprendizaje de programación* a partir de una implementación de Gobstones sobre un *editor proyectivo*, haciendo uso de sus cualidades intrínsecas para facilitar al alumno la comunicación de soluciones en términos de conceptos en lugar de trabajar sobre texto crudo, reduciendo así elementos superfluos que pudieran entorpecer la secuencia didáctica.

1. Introducción

En la República Argentina la enseñanza de la programación en el segundo ciclo primario y primer ciclo secundario se plantea utilizando lenguajes eminentemente visuales [8]. Esto permite a los alumnos concentrarse en aquello que desean expresar (es decir, el programa que pretenden crear) al eliminar ciertas dificultades inherentes en lenguajes sobre soporte de texto. Por ejemplo, el lenguaje Scratch¹[15] únicamente permite construcciones *sintácticamente válidas*, ya que cada comando del lenguaje es conformado por bloques visuales encastrables, de tal manera que dos bloques solo encastran cuando constituyen una combinación válida.

Luego, cuando el alumno pasa a un ciclo superior secundario o a la universidad, se le presentan lenguajes basados en texto, en los cuales los errores de sintaxis y de tipado son posibles. En particular en la Universidad Nacional de Quilmes, en la materia de Introducción a la Programación, se utiliza el lenguaje Gobstones [5], creado específicamente para la enseñanza de programación.

Si bien Gobstones cuenta con un modelo acotado y una secuencia didáctica clara que lo convierten en una gran herramienta para la enseñanza²; al ser basado en texto presenta un nivel de complejidad que puede no resultar adecuado para ciclos inferiores.

En este contexto surgió la propuesta de crear una implementación de Gobstones que permitiera únicamente sintaxis válida, de manera similar a los lenguajes visuales ya usados en la enseñanza de la programación, reduciendo así elementos superfluos que pudieran entorpecer la secuencia didáctica planteada para el segundo ciclo de la educación primaria y el primer ciclo de la educación secundaria. Además se presenta a esta implementación de Gobstones en un entorno que acompañe su secuencia didáctica, tanto desde la construcción de ejercicios y planteo de problemas, como la inclusión incremental de conceptos nuevos durante el proceso de aprendizaje. Se pretende entonces lograr una continuidad entre la enseñanza de la programación utilizando componentes visuales y texto, volviendo más gradual la transición entre uno y otro.

Para lograr esta experiencia cercana al texto pero con la ausencia de errores sintácticos, se decidió trabajar sobre un *editor proyectivo*, ya que solamente permite ingresar construcciones sintácticamente válidas en un

¹Lifelong Kindergarten Group. *Scratch*. URL: <https://scratch.mit.edu/>.

²Pablo E. Martínez López. *Por qué gobstones es importante*. URL: <http://gobstones-es-importante.blogspot.com.ar/>.

formato estandarizado (espacios, indentación y demás elementos estéticos son dados por el editor, no por el usuario).

1.1. Gobstones

Gobstones constituye un lenguaje y una secuencia didáctica diseñados para la enseñanza de la programación informática, orientada a personas con poca o nula experiencia previa. Propone al programa como una descripción denotacional de un conjunto de ideas que resulta ser ejecutable. Todo programa Gobstones trabaja sobre un modelo concreto constituido por un tablero de celdas cuadradas, sobre las cuales pueden depositarse o retirarse abalorios de colores utilizando un brazo mecánico que puede moverse por el tablero. El input del programa será un tablero inicial y el output será una transformación sobre ese tablero. De esta manera, el programa aplica una transformación que utiliza primitivas del tablero (poner bolita de color, mover cabezal del brazo mecánico, quitar bolita, etc.) y conceptos fundamentales de programación estructurada como procedimientos, funciones, repeticiones, alternativas condicionales, entre otros. La secuencia didáctica de gobstones describe un orden lógico para la presentación de los conceptos fundamentales, comenzando con procedimientos y parametrización, haciendo foco en la necesidad de dividir en subtarefas y modularizar. En la Fig.1 puede observarse un ejemplo de programa junto a su tablero resultante.

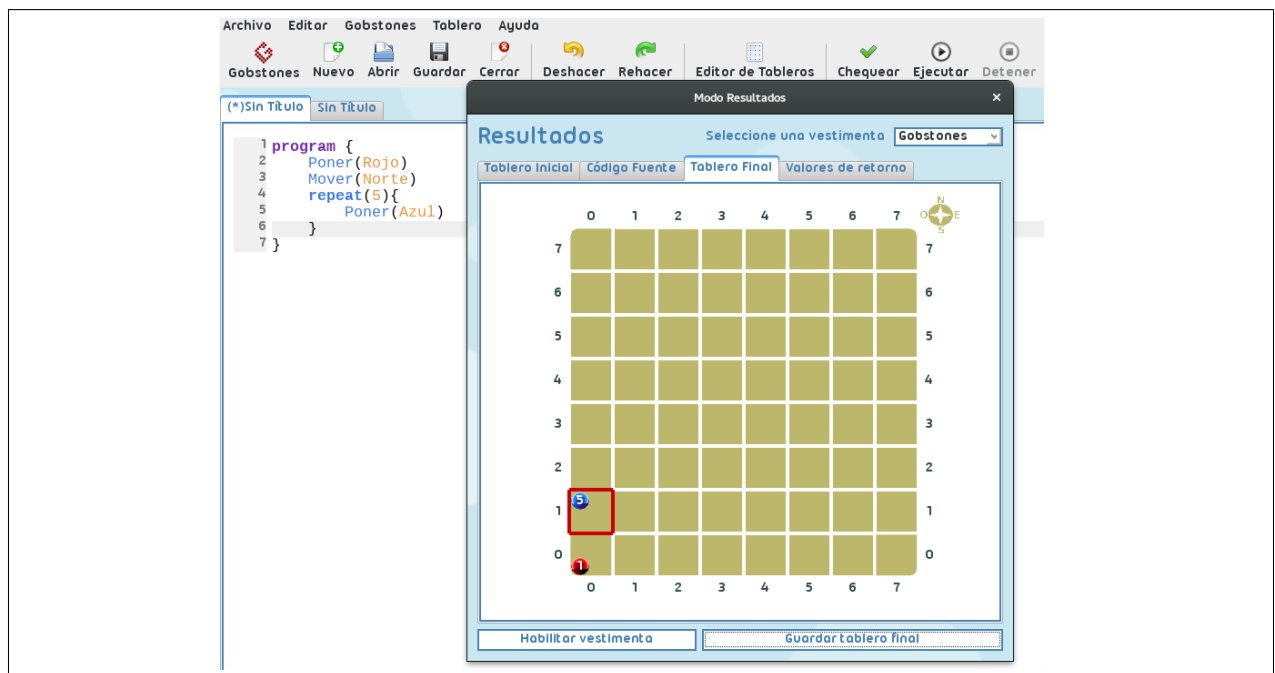


Figura 1: Ejemplo de un programa Gobstones, en el IDE PyGobstones³, con su respectivo tablero final

1.2. Editores Proyectivos

El término Editor Proyectivo fue acuñado por Martin Fowler⁴ en el año 2005, al intentar plantear un ambiente de desarrollo donde el programador pudiera expresar sus ideas en términos de conceptos en lugar de texto. Lo que vemos como texto pasaría entonces a constituir una representación editable del concepto al que hace referencia (y al cual Fowler llama *representación abstracta*).

De esta manera, los conceptos del lenguaje son el dominio de los editores proyectivos, y decimos que un programa es una *representación abstracta* construida utilizando dichos conceptos. Para modificar esta

⁴M. Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* URL: <http://martinfowler.com/articles/languageWorkbench.html>.

representación el programador interactúa con una interfaz de usuario, llamada *representación editable*, sobre la cual la *representación abstracta* se *proyecta* en forma de texto [7].

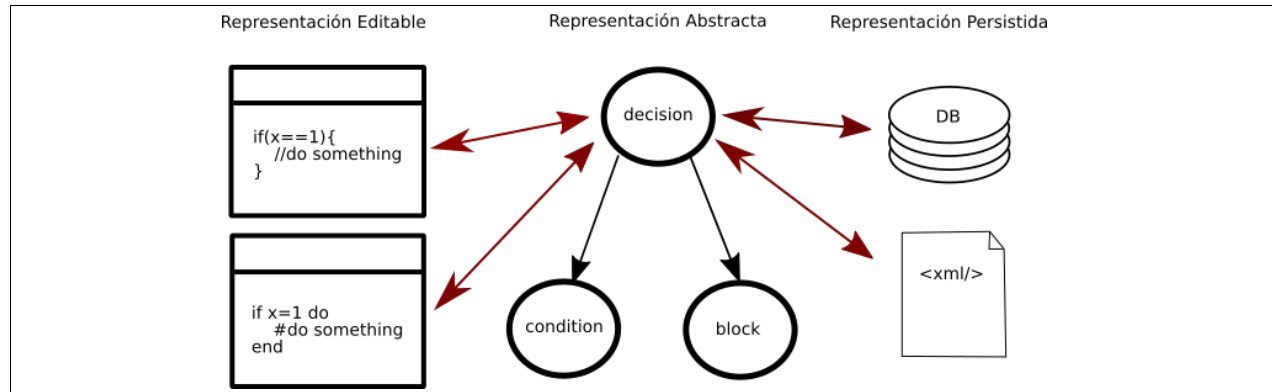


Figura 2: Representaciones de un programa en un editor proyectivo

A su vez, la *representación abstracta* puede persistirse de diferentes maneras a diferentes soportes, con lo cual se introduce la idea de *representación persistida* para hablar del formato en que guardará el programa, ya sea en una base de datos, un archivo binario, un texto con formato XML, etc.

Algunas consecuencias de los editores proyectivos:

- Deja de necesitarse un parser para el lenguaje, volviéndolo más sencillo de extender.
- El programador trabaja más cerca de los conceptos que quiere expresar.
- Reduce drásticamente los posibles errores de sintaxis.
- El editor trabaja directamente con las instancias de los conceptos, con lo cual:
 - es más sencillo analizar el programa
 - se simplifica la construcción de herramientas (ej: refactors, migrado de versiones de lenguaje, intentions, etc)
 - se mejora la performance del editor al eliminarse la etapa de parseo.

1.3. Organización de este informe

En la sección 2 se presenta la tecnología a usar y en la sección 3 se describe el desarrollo del modelo conceptual del lenguaje Gobstones en términos de esa tecnología. En la sección 4 se muestra cómo este modelo conceptual se proyecta sobre el editor. Una vez creado el editor, se procede a implementar el intérprete del lenguaje y la renderización de los tableros inicial y final. Teniendo el lenguaje básico funcionando, se trabaja en la sección 6 sobre el sistema de chequeo de tipos, orientado a asistir al estudiante mediante mensajes de error legibles. En la sección 7 se analizan problemas típicos de los editores proyectivos y se busca mejorar la experiencia de usuario aplicando diferentes técnicas que facilitan una edición más familiar, es decir, más cercana a una experiencia de edición de texto. Luego en la sección 8 se extiende el proyecto agregando la capacidad de *definir ejercicios*, que constituye un lenguaje específico de dominio cuya finalidad es permitirle al docente plantear ejercicios, desde título y descripción hasta restricciones de features de lenguaje y análisis de código. En la sección 9 se destacan brevemente los conocimientos aplicados durante el desarrollo del proyecto y por último en la sección 10 se cierra el informe con una conclusión e ideas sobre el camino que el proyecto pudiera seguir a futuro.

2. Tecnología proyectiva a usar

De los entornos proyectivos existentes hoy en día, se decidió utilizar el workbench Meta Programming System (MPS)⁵ de la empresa JetBrains, en su versión 3.3. Se trata de un entorno orientado al desarrollo de lenguajes maduro y estable, sobre el cual se realizaron exitosamente diferentes proyectos, entre los cuales se cuentan:

- MetaR⁶: un IDE que utiliza el lenguaje R para facilitar el análisis de datos biológicos.
- mbeddr⁷: un IDE orientado a la programación sobre hardware, que extiende el lenguaje C y soporta verificación formal, máquinas de estado y variabilidad en líneas de productos, entre otros.
- YouTrack⁸: un gestor de proyectos.

MPS brinda un DSL (lenguaje específico de dominio) para la definición de conceptos puros del lenguaje a implementar, y sobre estos la posibilidad de describir cómo este modelo se renderizará, comportamiento específico para cada concepto, sistema de tipos, etc. Al ser todas estas incumbencias transversales a los conceptos, se organizan en forma de aspectos. Los conceptos se comportan de manera similar a una clase en programación orientada a objetos, en tanto y en cuanto admiten extensión por herencia e implementación de interfaces. A su vez, estas construcciones determinan las instancias de nodos que compondrán un programa, comparables a los nodos de un árbol de sintaxis abstracta.

A partir de los lenguajes definidos en esta herramienta es posible generar un entorno de desarrollo autónomo o plugins para entornos pre-existentes.

3. Modelo conceptual del lenguaje Gobstones

Se comenzó modelando el lenguaje Gobstones en términos de conceptos. Como puede observarse en el ejemplo de la *Fig. 3*, para cada concepto pueden definirse sus posibles nodos hijo, propiedades y referencias a otros nodos. Se organizan en una jerarquía, pudiendo extender de otros conceptos e implementar interfaces. En este caso, el concepto *IfElseStatement* extiende del concepto abstracto *Statement*, y sus posibles hijos son una expresión, que será la condición de la alternativa, un bloque de sentencias para el caso en que la condición sea verdadera, y otro bloque de sentencias para el caso contrario.

```
concept IfElseStatement extends Statement
                                implements <none>

instance can be root: false
alias: if
short description: Condicional

properties:
<< ... >>

children:
condition      : Expression[1]
ifTrueBlock    : StatementList[1]
ifFalseBlock   : StatementList[1]

references:
<< ... >>
```

Figura 3: Definición del concepto para la alternativa condicional

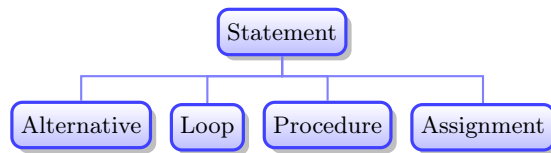
⁵JetBrains s.r.o. *MPS Workbench*. URL: <https://www.jetbrains.com/mps/>.

⁶Campagne Laboratory. *MetaR*. URL: <http://metaR.campagnelab.org>.

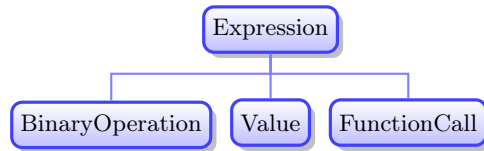
⁷itemis y fortiss. *mbeddr*. URL: <http://mbeddr.com/>.

⁸JetBrains s.r.o. *YouTrack*. URL: <http://www.jetbrains.com/youtrack/>.

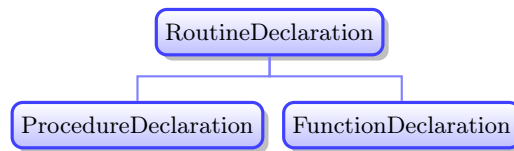
Los conceptos más importantes son *Statement*, que denota un comando, y *Expression* que denota una expresión que puede ser evaluada. De manera simplificada, tenemos que el primer nivel de la jerarquía de sentencias quedó dado por:



Y el primer nivel de la jerarquía de expresiones se compone de:



Además, se tiene una jerarquía separada para la definición de rutinas:



Decimos entonces que un programa Gobstones básico, en el contexto de este editor proyectivo, se encuentra dado por una colección de sentencias y una colección de definición de rutinas.

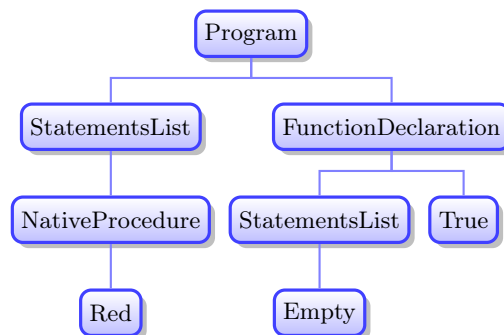
Donde el siguiente programa:

```

program {
  Poner ( Rojo )
}

function verdadero () {
  return ( True )
}
  
```

Corresponde a un modelo conceptual con la estructura:



4. Implementación del Editor

Una vez completado el modelo conceptual con los elementos del lenguaje Gobstones, según la especificación de *Las bases conceptuales de la Programación: Una nueva forma de aprender a programar* [5], se procedió a definir el aspecto de editor para cada uno de ellos. Para esto se hace uso de otro DSL provisto por MPS, que permite definir los layouts en los cuales los conceptos se renderizarán.

La finalidad de este editor es proveer una interfaz de usuario que brinde una experiencia similar a la edición de texto, pero permitiendo únicamente construcciones válidas. Para lograr esto, internamente se hace una distinción entre nodos ligados al programa, es decir, nodos que representan conceptos que se escribieron exitosamente y forman parte del programa que se está escribiendo, y nodos en proceso de creación. Cuando el usuario comienza a editar un nodo, internamente se crea una instancia de un editor que se encarga de crear el nodo correspondiente a aquello que se está escribiendo, siempre y cuando logre relacionarlo con un concepto válido. De esta manera, funcionalidades como el autocompletado son responsabilidad de la instancia del editor, mientras que otras como los estilos, sugerencias, alertas, etc. son aplicadas sobre las instancias de los nodos de cada concepto.

```
<default> editor for concept FunctionDeclaration
node cell layout:
  [-
  # alias # { name } ( ( - % arguments % /empty cell: <default> - ) ) {
  ( - % statements % /empty cell: - )
  return ( % return % )
  }
  /folded cell: [> # alias # { name } <]
  -]

inspected cell layout:
<choose cell model>
```

Figura 4: Definición del aspecto de edición para el concepto de declaración de función

En la Fig. 4 se observa el aspecto de edición definido para la declaración de funciones. En este quedan relacionadas las propiedades e hijos del concepto *FunctionDeclaration*.

Esta definición provee un *binding bidireccional* entre el modelo conceptual y el editor, comparable con un patrón de arquitectura MVP⁹¹⁰. Como consecuencia, cualquier cambio en el modelo se verá reflejado instantáneamente en el editor, y cualquier cambio realizado desde el editor modificará los nodos del modelo conceptual; habilitando también la posibilidad de tener diferentes vistas para un mismo modelo, cuya utilidad se verá en la sección siguiente, al lograr dos vistas diferentes para un mismo modelo de tablero.

```
<default> editor for concept BinaryOperation
node cell layout:
  [- % left % # alias # % right % -]

inspected cell layout:
<choose cell model>
```

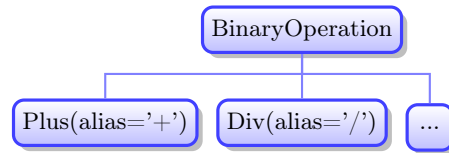
Figura 5: Definición del aspecto de edición para el concepto abstracto de operación binaria

El diseño del editor buscó respetar los principios de la programación orientada a objetos. Un claro ejemplo son los subconceptos de *BinaryOperation* tales como los operadores lógicos y aritméticos, que reutilizan una misma vista. Para lograr esto se extrapola el clásico patrón *Template Method* [2], donde el aspecto de edición

⁹Martin Fowler. *GUI Architectures*. URL: <http://martinfowler.com/eaDev/uiArchs.html>.

¹⁰Microsoft. *Data Binding (WPF)*. 2014. URL: [http://msdn.microsoft.com/en-us/library/ms750612\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms750612(v=vs.110).aspx).

del concepto abstracto *BinaryOperation* define el layout de Fig.5, y cada subconcepto implementa un alias distinto, correspondiente con el símbolo de su operación.



Por otro lado, el coloreado de sintaxis, indentación, retorno de carro y demás detalles estéticos se determinan mediante una planilla de estilos, cuidando mantener consistencia entre los diferentes elementos del lenguaje. Por ejemplo, si bien la implementación interna de llamadas a procedimientos nativos y llamadas a procedimientos de usuario son diferentes, ambos comparten los mismos estilos Fig.6.

Style:

```

<no base style> {
  text-foreground-color : darkMagenta
}

```

Common:

cell id	<default>
action map	<default>
keymap	<default>
menu	<none>
attracts focus	noAttraction
show if	<no condition>

Property cell:

property	name
text*	<none>
empty text*	false
read only	true
allow empty	false

Figura 6: Definición de estilos para la invocación de procedimientos

Se puede apreciar en Fig.7 que el editor resultante incluye funcionalidades como:

- **Autocompletado inteligente:** solo autocompleta con elementos coherentes con el contexto. En el ejemplo solo autocompleta con sentencias y no expresiones, ya que estas últimas no cumplen propósito alguno en el cuerpo del procedimiento.
- **Coloreado:** Se colorean palabras claves, identificadores, etc. Nótese que los procedimientos y funciones mantienen el mismo color tanto en la declaración como en la invocación, y se diferencian las rutinas nativas de las creadas por el usuario con el uso de negrita en la fuente.

- **Colapsado de bloques:** se pueden colapsar procedimientos y funciones, con el objetivo de facilitar la legibilidad del programa como un todo.
- **Autoindentación:** Las sentencias que se agregan son indentadas automáticamente.
- **Plantilla de cada herramienta:** Al comenzar a escribir una estructura de control, la misma se crea por completo, y permite al usuario rellenar sus partes, proveyendo información sobre qué tipo de elementos corresponden a cada parte. En la imagen, puede notarse esto en la rama falsa de la alternativa condicional.

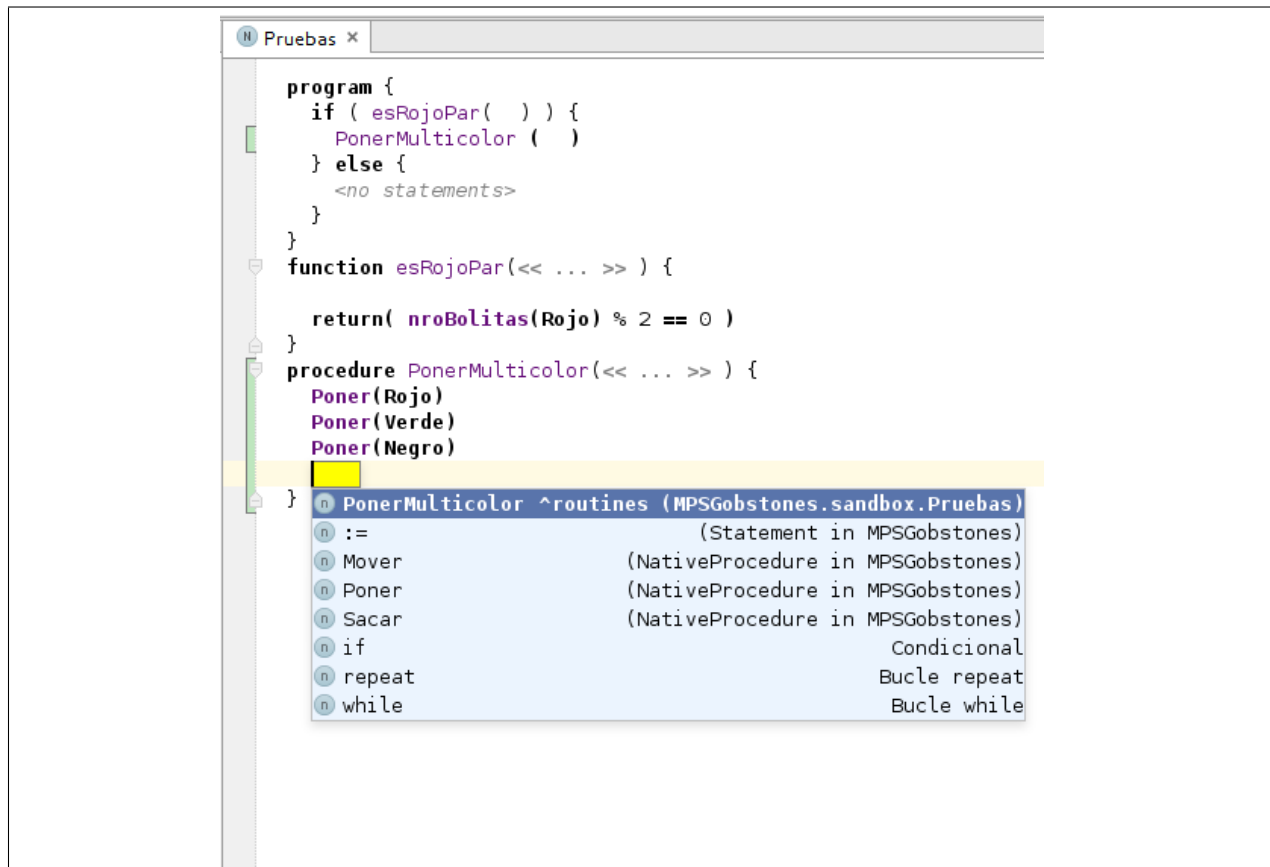


Figura 7: Editor proyectivo de Gobstones

5. Implementación del Intérprete

Una vez definido el editor proyectivo debemos ser capaces de ejecutar el programa. En esta instancia se presentaron varias alternativas de implementación posibles: compilando o traduciendo a otro lenguaje que luego será ejecutado, es decir, generando de código; o interpretando el programa sin generar código. La generación de código hubiera implicado depender de un compilador o una VM específica, y por consiguiente complejizado la instalación del producto final (ya sea dependiendo de que la máquina donde se fuera a usar la herramienta final poseyera este compilador, o al verse obligado a empaquetar un compilador o implementación de VM junto con el instalador del entorno), con lo cual se optó por desarrollar un intérprete a partir del ambiente de nodos de conceptos.

En esta etapa se hizo uso del aspecto de comportamiento (*behaviour*), que permitió agregar métodos e inicializaciones a los conceptos, utilizando un DSL con una sintaxis similar a Java. Esto fue así porque al momento de implementar el intérprete, la herramienta no contaba con un aspecto dedicado exclusivamente a la interpretación. Sin embargo, futuras versiones de MPS permitirán extensión mediante la definición de nuevos aspectos por parte del usuario, con lo cual una práctica recomendable será encapsular el comportamiento del intérprete en un aspecto distinto del de *behaviour*, y que podremos llamar *interpreter*.

Se definió en el concepto abstracto *Statement* un método *interpret* que toma como argumento un estado del programa, y retorna otro estado del programa. Por defecto la implementación de *interpret* retorna el estado sin cambiarlo, como se observa en Fig.8. En otras palabras, las sentencias quedaron implementadas un patrón *Composite* [2] en el cual cada sentencia realiza su transformación sobre el estado y delega la interpretación en sentencias hijas de ser necesario. Por ejemplo, en la Fig.9 se puede observar cómo la interpretación de la alternativa condicional está dada por una evaluación de su condición, seguida de la interpretación de alguno de sus bloques según esa condición haya resultado verdadera o falsa.

```
concept behavior Statement {

    constructor {
        <no statements>
    }

    public virtual InterpreterState interpret(InterpreterState state) {
        return state;
    }
}
```

Figura 8: Comportamiento del concepto abstracto Statement

```
public InterpreterState interpret(InterpreterState state)
overrides Statement.interpret {
    boolean condition = ((boolean) this.condition.reduce(state).value);
    (condition ? this.ifTrueBlock : this.ifFalseBlock).interpretEach(state);
    return state;
}
```

Figura 9: Interpretación de la alternativa condicional

La evaluación de esta condición es posible porque, de manera similar a como se implementa la interpretación para cada sentencia, cada *Expression* entiende el mensaje *reduce*, que toma como argumento un estado del programa y retorna una instancia de *InterpreterValue*, la cual encapsula el valor resultante y contiene información del tipo de este valor.

5.1. Contexto de ejecución

El estado del programa que es necesario para interpretar sentencias y evaluar expresiones quedó determinado por dos pilas: una pila de tableros (instancias de la clase de dominio *Board*) y una pila de contextos, donde se entiende como contexto al conjunto de variables a las que se pueden acceder en un momento dado.

Al implementar rutinas se siguió la especificación de gobstones al elegir una evaluación *Call by Value* [3] de los argumentos, lo cual a su vez permitió mantener una implementación sencilla, evaluando primero los argumentos de las rutinas antes de crear el nuevo contexto. Cada vez que se invoca un procedimiento, el contexto que se estaba usando se apila y se crea un nuevo contexto con variables asociadas al resultado de evaluar cada argumento, si los hubiere.

Cabe aclarar que de haberse requerido una implementación *Call By Name* podría haberse llevarse a cabo inicializando las variables en el nuevo contexto con expresiones asociadas al contexto donde fueron definidas, en lugar de asociarlas a valores puntuales. Es decir, en lugar de tener un *InterpreterValue*, se

podría haber creado una *ContextAwareExpression*, que únicamente se evaluaría en caso de ser requerida. La complejidad extra de esta implementación habría radicado en controlar los posibles *memory leaks* que se pudieran generar al mantener referencias a diferentes contextos y guardar el resultado de la primer evaluación para no recalcularlo en evaluaciones posteriores.

Volviendo a la implementación actual, para el caso de las funciones no solo se debe apilar el contexto de variables, sino que también debe clonarse el estado existente del tablero y apilarse, ya que en Gobstones las funciones no realizan efectos de lado, sino que todo efecto aplicado al tablero desaparece una vez que la función retorna un valor. Esto queda claro en *Fig.10*, donde la invocación de función primero evalúa sus argumentos, crea un *IsolatedContext* con esos argumentos inicializados, y luego evalúa su cuerpo dentro de ese contexto.

```
public InterpreterValue reduce(InterpreterState state)
    overrides Expression.reduce {
    map<string, InterpreterValue> parameterVariables = initializeParameterVariables(state);
    state.startIsolatedContext();
    state.context.putAll(parameterVariables);
    foreach statement in this.declaration.statements {
        statement.interpret(state);
    }
    InterpreterValue result = this.declaration.return.reduce(state);
    state.endIsolatedContext();
    return result;
}
```

Figura 10: Interpretación y cambio de contexto en la invocación de una función

5.2. Clases de dominio de Gobstones

El proyecto se organiza en varios subproyectos o módulos: un proyecto para el editor, otro para definir el tablero como componente customizado, y otro más para el modelo puro de Gobstones. En este último se definieron las entidades que constituyen el dominio de Gobstones y su comportamiento, tales como el tablero, las celdas, las bolitas, entre otros, utilizando una implementación proyectiva del lenguaje Java. En este modelo quedaron definidas las funcionalidades nativas que le permiten al intérprete interactuar con el tablero de forma tal que, por ejemplo, la interpretación del procedimiento *Poner(<color>)* se termina traduciendo en un mensaje *board.addStones(Color color, int quantity)*. En otras palabras, toda la lógica perteneciente a los elementos concretos de Gobstones queda encapsulada en estas clases de dominio, en su módulo específico e independiente de los otros módulos. Estas clases constituyen una de las porciones de código más ejecutadas del proyecto y requirieron el uso de estructuras de datos adecuadas para evitar un impacto notable en la performance, no sólo en términos de procesamiento, sino también en generación de instancias. En particular, el tablero guarda la información de sus celdas en un *TreeSet*¹¹ para únicamente generar la cantidad de instancias de celdas necesaria. Si bien otras optimizaciones más agresivas eran posibles, no surgió la necesidad de aplicarlas y muchas hubieran implicado reducir la calidad del diseño trabajando sobre estructuras de más bajo nivel.

5.3. Edición del tablero inicial

Para editar el tablero inicial que constituye el input del programa Gobstones se creó un editor específico con un *layout table*, como se observa en la *Fig.11*. Este tablero se utiliza para inicializar la pila de tableros del estado del programa y puede definirse en un archivo virtual separado de aquel que contiene el código fuente.

La filosofía detrás de este diseño es que cada componente del programa, desde su tablero inicial y su código hasta su tablero final, son representados como archivos. Esto ayuda a mantener una interfaz de usuario minimalista y una forma de interacción consistente para con los diferentes elementos del entorno.

¹¹Oracle. *TreeSet*. URL: <https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html>.

5.4. Renderización del tablero final

El tablero final es una vista con *binding unidireccional* que renderiza el estado del tablero que resulta de interpretar el programa Gobstones. Esta vista no fue construida con componentes pre-existentes en MPS, sino que se extendió el editor de MPS creando un componente de Swing¹² que toma un *Board* y lo renderiza, como se aprecia en la *Fig.12*. Nótese que tanto el tablero inicial como el tablero final son vistas del modelo *Board*, y se usa una u otra según el tipo de interacción que se desee permitir.

6. Chequeo de tipos

Para implementar el chequeo de tipos se tomó como base el trabajo de inserción profesional de *Federico A. Sawady O'Connor* titulado *Formalización e implementación de un sistema de tipos para el lenguaje de programación Gobstones* [4], donde se presentan las reglas de inferencia de tipos necesarias para implementar los chequeos de tipos en Gobstones. Tomando las reglas puras como base, se implementaron las reglas en MPS dentro del aspecto de sistema de tipos *typesystem*. Por ejemplo, la regla de tipado de las operaciones aritméticas queda denotada por *typeof_IntegerOperation* en *Fig.13*. En la *Fig.14* se ejemplifica un error durante el chequeo de tipos, el cual se muestra como un error legible en castellano.

Además de las reglas de inferencia, es posible crear chequeos manuales por fuera del sistema de tipos. Esto fue de utilidad a la hora de validar que los nombres de procedimientos comiencen con mayúscula y los nombres de funciones con minúscula, ya que al no existir lexers ni parsers, este tipo de restricciones deben declararse en una regla propia.

7. Mejorando la experiencia de Usuario

Mantener una experiencia cercana al texto fue uno de los mayores desafíos durante el desarrollo. Por defecto, la edición requiere una escritura en orden prefijo, ya que, por ejemplo, cada operador es padre de sus operandos, y es necesario primero definir la instancia del nodo padre antes de poder determinar sus hijos. Es decir, al escribir $1 + 2$ primero es necesario escribir el $+$ y luego se generará la estructura para completar los hijos de izquierda y derecha $_ + _$. Sin embargo, esta notación puede resultar incómoda y particularmente antinatural en un curso introductorio a la programación, por lo cual surge la necesidad de permitir una notación infija.

La implementación de notación infija se llevó a cabo declarando acciones de transformación y reemplazo de nodos, para las cuales se declararon sus condiciones disparadoras. Volviendo al ejemplo de las operaciones aritméticas (o, en general, de operadores binarios), cuando se comienza escribiendo:

¹²Oracle. *Swing*. URL: <http://docs.oracle.com/javase/7/docs/technotes/guides/swing/>.



Figura 11: Tablero inicial

Resultado del programa: Introducción a Gobstones				
0:0 2	1:0	2:0	3:0	4:0
0:1	1:1	2:1	3:1	4:1
0:2	1:2	2:2	3:2	4:2
0:3	1:3	2:3	3:3	4:3

Figura 12: Tablero final

```

rule typeof_IntegerOperation {
  applicable for concept = IntegerOperation as integerOperation
  overrides true

  do {
    node<IntegerType> integer = new node<IntegerType>();
    infer typeof(integerOperation.left) :<=: integer;
    infer typeof(integerOperation.right) :<=: integer;
    typeof(integerOperation) :==: integer;
  }
}

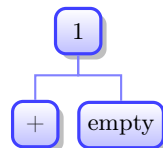
```

Figura 13: Regla de inferencia de tipos para operaciones aritméticas



normalmente el editor buscaría reconocer al + como parte de cierto identificador o como posible hijo de 1, pasando a un estado de error, como se ejemplifica en *Fig.15*

Al definir una acción de reemplazo, fue posible identificar esta ocurrencia mediante una expresión regular que reconoce que se escribió un número seguido de un alias de un operador. Luego, se reemplaza la construcción:



Por

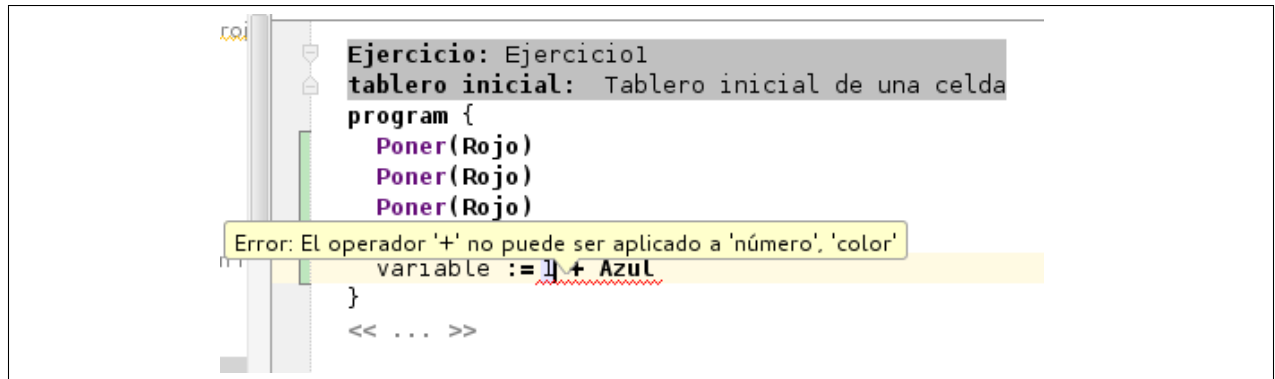


Figura 14: Ejemplo de alerta durante el chequeo de tipos

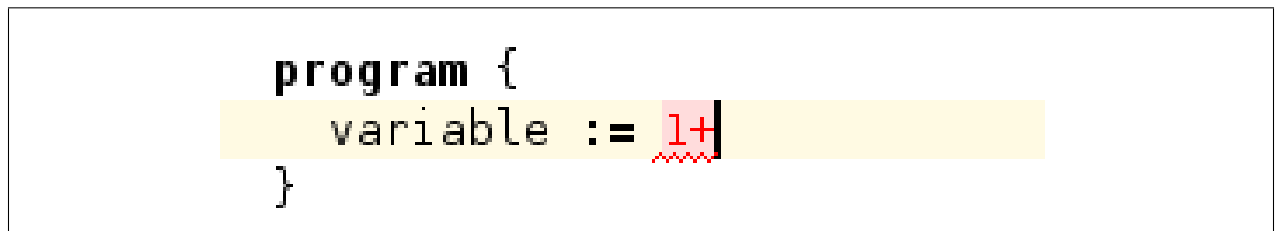
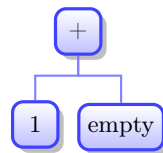


Figura 15: Edición de operaciones binarias sin acciones de reemplazo



De manera automática y sin que el usuario lo note, mostrándole *Fig.16*

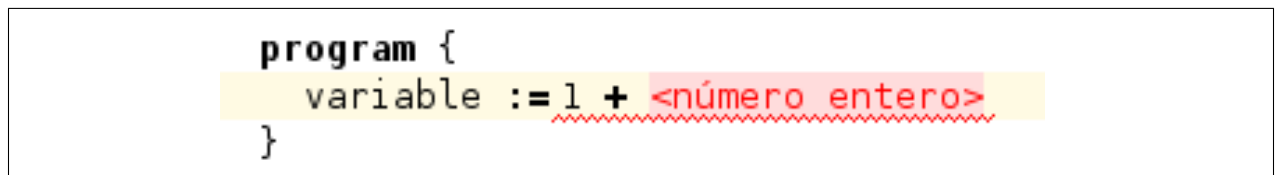
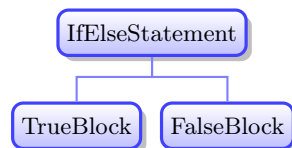


Figura 16: Edición de operaciones binarias con acciones de reemplazo definidas

Algo similar ocurre con la alternativa condicional, pero esta vez cuando el usuario intenta borrar la rama falsa y dejar únicamente la verdadera. Al escribir el *if* se autogenera la plantilla correspondiente *Fig.17*

pero si se intentaba borrar la rama del *else*, se terminaba borrando toda la construcción del *if*. Esto sucedía porque el borrado se realiza por nodo, y la alternativa es toda un único nodo. Para evitar este problema, se volvió optativa la rama *false* y se definió una acción que se dispare al intentar borrar un *if*, de tal manera que en lugar de borrarlo se elimine su hijo:



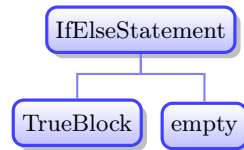
Por otra construcción que mantiene el mismo hijo *TrueBlock* y el padre, pero se borra el hijo *FalseBlock*:

```

program {
  if (            ) {
    <no statements>
  } else {
    <no statements>
  }
}

```

Figura 17: Editor de la alternativa condicional con dos ramas



8. Definición de Ejercicios

Hasta el momento, las implementaciones existentes de gobstones carecían del concepto de proyecto como unidad de organización, y sólo se podía tener un archivo fuente principal y una única biblioteca. Sin embargo, esta única biblioteca comenzó a tener dos usos bien diferenciados por parte de docentes y alumnos: por un lado el alumno quería tener siempre disponible rutinas reutilizables que había hecho para otros ejercicios, y por el otro un docente podía querer brindarle al alumno una biblioteca con funciones y procedimientos ya creados, para trabajar sobre un ejercicio en un nivel de abstracción más alto. Viendo esta dicotomía, se decidió agregar la idea proyecto bajo el rótulo de *ejercicio*, con la intención de permitir no solo declarar enunciados de ejercicios, sino también de poder proveer al alumno de bibliotecas diseñadas por el docente (si el ejercicio en cuestión lo amerita) separadas de las bibliotecas de alumno.

Estas bibliotecas no constituyen *imports* en el sentido tradicional, dado que no se establecen desde el programa, sino que son dependencias inyectadas por el ejercicio. Esto permite agregar bibliotecas sin modificar el lenguaje Gobstones, y por ende sin aumentar la complejidad del programa creado por el alumno. Por el momento no hay implementada una resolución de posibles conflictos de nombres entre diferentes bibliotecas, quedando a discreción de los usuarios.

Siguiendo la filosofía planteada en este proyecto, la definición del ejercicio también se visualiza como un archivo, orientado a ser legible y con apariencia de documentación, como se observa en la *Fig.18*

A su vez, dentro de la definición del ejercicio es posible declarar qué herramientas del lenguaje no pueden ser usadas dentro de la resolución del mismo, como muestra la *Fig.19*

Internamente, todo concepto del lenguaje que implemente la interfaz *CanBeRestricted* aparecerá en la lista de conceptos cuyo uso puede prohibirse. Esto permite reforzar la introducción gradual de conceptos de programación: al presentarle al alumno un problema que no puede resolver con las herramientas que conoce, se genera la necesidad de indagar por aquello que le está faltando. A este tipo de aprendizaje se lo conoce como *aprendizaje basado en problemas* o *aprendizaje por indagación* [1], y constituye uno de los pilares fundacionales de la secuencia didáctica de Gobstones.

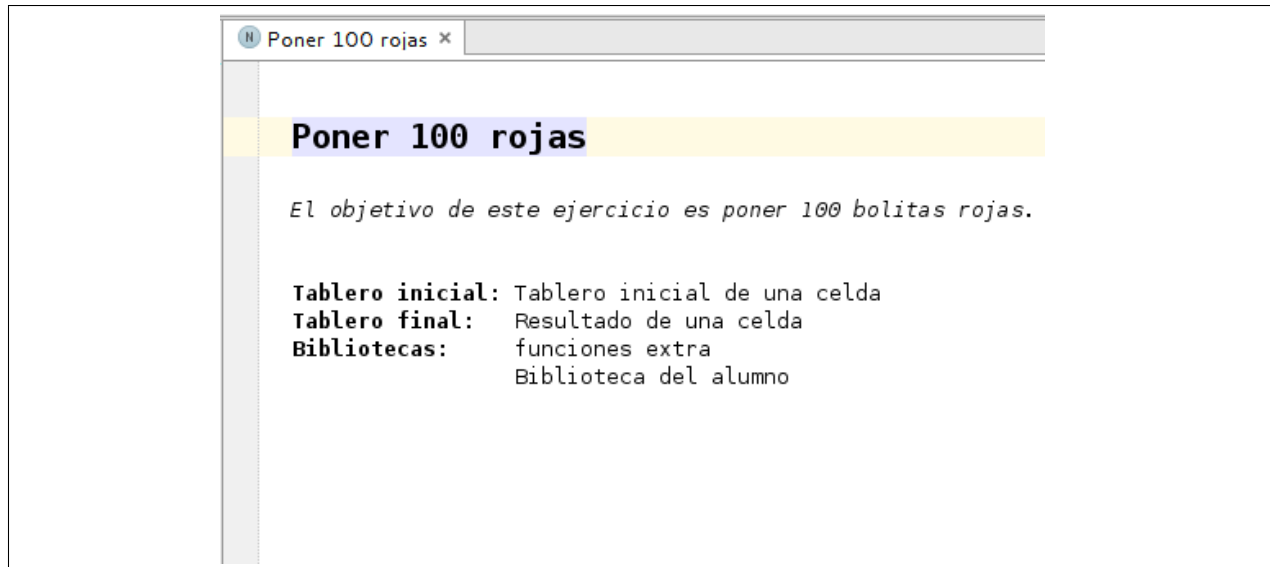


Figura 18: Definición de un ejercicio simple

9. Conocimientos Aplicados

El presente trabajo requirió la aplicación numerosos conocimientos técnicos y teóricos, entre los cuales se destacan:

- **Programación orientada a objetos:** se aplicó de manera transversal a todo el proyecto, desde el modelo puro y las vistas hasta la implementación de intérprete.
- **Teoría de lenguajes:** para el chequeo de tipos y diseño del intérprete.
- **Metaprogramación:** todo el proyecto es, en sí mismo, una aplicación de metaprogramación.
- **Construcción de interfaces de usuario:** fundametal para el diseño de las vistas, ya que al desaparecer la sintaxis y el texto puro, todo lo que ocupa su lugar es constituido por interfaces de usuario.

10. Conclusión

Se ha logrado finalizar exitosamente una primera versión usable de una implementación del lenguaje gobstones sobre un editor proyectivo. Los conceptos y tecnologías aplicados permitieron agregar numerosas funcionalidades esperables en un entorno de programación moderno sin grandes dificultades, y por su misma naturaleza requirieron prestar particular atención a ciertos problemas de usabilidad que devienen de la no utilización de texto. La capacidad de definir ejercicios con sus respectivas especificaciones, tableros, bibliotecas y restricciones constituye un aporte novedoso respecto de las implementaciones pre-existentes. Por este mismo motivo la utilidad de esta funcionalidad, así como también posibles modificaciones futuras, queda aún por ser demostrada y dependerá de la respuesta de los usuarios. Otras funcionalidades, especificadas en *Las bases conceptuales de la Programación* [5] y presentes en *PyGobstones*¹³, como la posibilidad de cambiar vestimentas (estilos e imágenes) a los tableros, han resultado ser sumamente útiles a la secuencia didáctica de Gobstones y se proyecta agregarlas en versiones futuras del producto. Debido a que actualmente el entorno no se expuso aún a una cantidad suficiente de usuarios y testeo manual, y no posee los estilos gráficos propios de Gobstones, no se considera que esté preparado para ser usado aún en un ambiente productivo.

¹³Gobstones. *Qué es Gobstones*. URL: http://www.gobstones.org/?page_id=50.

Poner 100 rojas

El objetivo de este ejercicio es poner 100 bolitas rojas.

Tablero inicial: Tablero inicial de una celda

Tablero final: Resultado de una celda

Bibliotecas: funciones extra
Biblioteca del alumno

Restricciones:

Está prohibido usar **function**

Está prohibido usar

- function (BaseConcept in MPSGobstones)
- if Condicional
- procedure (BaseConcept in MPSGobstones)
- repeat Bucle repeat
- while Bucle while

Figura 19: Definición de ejercicio con restricciones

11. Anexo

Repositorio del proyecto: <https://github.com/uqbar-project/projectional-gobstones>

12. Agradecimientos

Mi más sincero agradecimiento a los numerosos docentes y compañeros, muchos de ellos potenciales usuarios, que me brindaron invaluable consejos y opiniones, animándome a continuar con un producto que espero podrán disfrutar en un futuro cercano.

Referencias

- [1] Joseph J. Schwab. «Inquiry, the Science Teacher, and the Educator». En: *The School Review* 68.2 (1960), págs. 176-195. ISSN: 00366773. URL: <http://www.jstor.org/stable/1083585>.
- [2] Erich Gamma y col. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.
- [3] Gilles Dowek y Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Undergraduate Topics in Computer Science. Springer, 2011. ISBN: 978-0-85729-075-5. DOI: 10.1007/978-0-85729-076-2. URL: <http://dx.doi.org/10.1007/978-0-85729-076-2>.
- [4] Federico A. Sawady O'Connor. «Formalización e implementación de un sistema de tipos para el lenguaje de programación Gobstones». En: (2012).
- [5] Pablo E. Martínez López. *Las bases conceptuales de la Programación: Una nueva forma de aprender a programar*. La Plata, Buenos Aires, Argentina, 2013. ISBN: 978-987-33-4081-9. URL: <http://www.gobstones.org/bibliografia/Libros/BasesConceptualesProg.pdf>.
- [7] Markus Voelter y col. «Towards User-Friendly Projectional Editors». En: *7th International Conference on Software Language Engineering (SLE)*. 2014.
- [8] Federico Sawady O'Connor Pablo Factorovich. *Actividades para aprender a Program.AR*. 2015. ISBN: 978-987-27416-1-7.
- [15] John Maloney y col. «Scratch: A Sneak Preview». En: *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing, IEEE Computer Society*, págs. 104-109.