Propuesta de 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

Propuesta de Trabajo de Inserción Profesional [1]

***Resumen*—El lenguaje Gobstones posee una secuencia didáctica bien definida que ha demostrado ser eficaz tanto en cursos iniciales universitarios como en escuelas secundarias. En el marco de una comunidad creciente de usuarios, proponemos el desarrollo 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 facilitarle al alumno la comunicación de soluciones en términos de conceptos en lugar de trabajar sobre texto plano, reduciendo así elementos superfluos que pudieran entorpecer la secuencia didáctica.**

I. 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 [6]. 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 Scratch[9][12] 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[3], 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 enseñanza universitaria y de ciclos superiores del secundario; al ser basado en texto presenta un nivel de complejidad que puede no resultar adecuado para el segundo ciclo de la educación primaria o el primer ciclo de la educación secundaria.

En este contexto surge la propuesta de crear una implementación de Gobstones que permita 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 paulatina de conceptos nuevos durante el proceso de aprendizaje. Se pretende 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, resulta idóneo el uso de un Editor Proyectivo, término acuñado por Martin Fowler en el año 2005[7] al intentar plantear un ambiente de desarrollo donde el programador pueda 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*). 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 representación el programador interactúa con una interfaz de usuario, llamada *representación editable*, sobre la cual la *representación abstracta* se *proyecta* en forma de texto[5].

De esta manera, el editor solamente permite ingresar construcciones sintácticamente válidas en un formato estandarizado (espacios, identación y demás elementos estéticos son dados por el editor, no por el usuario).

En la sección *II* se presenta la tecnología a usar y en la sección *III* se describe el desarrollo del modelo conceptual del lenguaje Gobstones en términos de esa tecnología. En la sección *IV* 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 *V* sobre el sistema de inferencia de tipos, orientado a asistir al estudiante mediante mensajes de error legibles. En la sección *VI* 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 expriencia de edición de texto. Luego en la sección *VII* se extiende el proyecto agregando un *lenguaje de definición de ejercicios*, que constituye un lenguaje específico de dominio cuya finalidad es

permitirle al docente plantear ejercicios, desde título y descripción hasta restriciones de features de lenguaje y análisis de código. Por último en la sección *VIII* se cierra el informe con una conclusión e ideas sobre el camino que el proyecto pudiera seguir a futuro.

II. Tecnología proyectiva a usar

De los entornos proyectivos existentes hoy en día, se decide utilizar el workbench Meta Programming System (MPS)[14] 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:

- MetaR[11]: un IDE que utiliza el lenguaje R para facilitar el análisis de datos biológicos.
- mbeddr[10]: 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.
- YouTrack[15]: un gestor de proyectos.

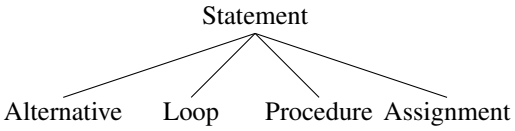
MPS brinda un DSL para la definición de conceptos puros del lenguaje a implementar, y sobre estos la posibilidad de describir cómo este modelo se renderizará, comportamiento específico para cada concepto, sistema de tipos, etc. Al ser todas estas incumbencias transversales a los conceptos, se organizan 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 IDE autónomo o pluguins para IDEs pre-existentes.

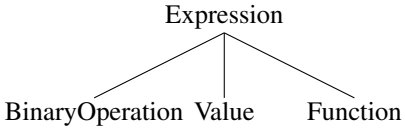
III. Modelo conceptual del lenguaje Gobstones

Se comienza modelando el lenguaje Gobstones en términos de conceptos. Como puede observarse en el ejemplo de *Fig.1*, 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, puede verse que 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.

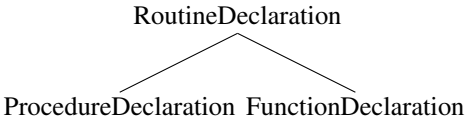
Los conceptos más importantes serán *Statement*, que denota un comando, y *Expression* que denota una expresión que puede ser evaluada. De manera simplificada, tenemos que el primer nivel de la jerarquía de sentencias queda dado por:



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 se encuentra dado por una colección de sentencias y una colección de definición de rutinas.

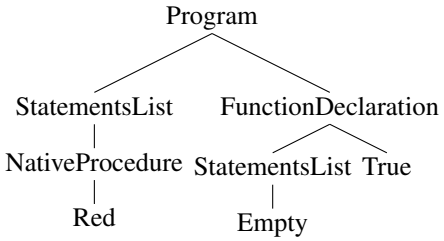
Donde el siguiente programa:

```
program {
  Poner(Rojo)
}

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

```
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 1. Definición del concepto para la alternativa condicional

Corresponde a un modelo conceptual con la estructura:



IV. 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*[3], se procede a definir el aspecto de editor para cada uno de ellos. Para ello, 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 2. Definición del aspecto de edición para el concepto de declaración de función

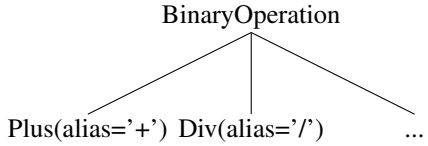
En la *Fig.2* se observa el aspecto de edición definido para la declaración de funciones. En este quedan mapeadas las propiedades e hijos del concepto *FunctionDeclaration*.

Esta definición provee un *binding bidireccional* entre el modelo conceptual y el editor, comparable con un patrón de arquitectura MVP [8][4]. 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.

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

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

Figura 3. Definición del aspecto de edición para el concepto abstracto de operación binaria

El diseño del editor busca 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*[1], donde el aspecto de edición del concepto abstracto *BinaryOperation* define el layout de *Fig.3*, y cada subconcepto implementa un alias distinto, correspondiente con el símbolo de su operación.



Por otro lado, el coloreado de sintaxis, identación, 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.

V. Implementación del Intérprete

Una vez definido el editor proyectivo, debemos ser capaces de ejecutar el programa. Una opción para llevar esto a cabo es compilar o traducir a otro lenguaje que luego será ejecutado, es decir, generando de código; otra alternativa es interpretar 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 hace uso del aspecto de comportamiento, que permite agregar métodos e inicializaciones a los conceptos, utilizando un DSL con una sintaxis similar a Java. Se define 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.4*

```
concept behavior Statement {

  constructor {
    <no statements>
  }

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

}
```

Figura 4. Comportamiento del concepto abstracto Statement

De esta manera, se tiene un patrón *Composite*[1] 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.5* 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.

```
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 5. 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.

V-A. Contexto de ejecución

El estado del programa que es necesario para interpretar sentencias y evaluar expresiones está dado 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 debió tomar la decisión entre una evaluación *Call By Name* o *Call by Value*[2] de los argumentos. Para mantener una implementación sencilla, se decidió utilizar *Call by Value*, 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 una implementación *Call By Name* podría 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 sólo se evaluaría en caso de ser requerida. Una problemática adicional de esta implementación radica en controlar los posibles *memory leaks* que se generan al guardar referencias a muchos 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 sólo 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.6*, 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 6. Interpretación y cambio de contexto en la invocación de una función

V-B. Clases de dominio de Gobstones

En un subproyecto separado del del editor, llamado *JavaGobstones*, se definen las clases de dominio de Gobstones: tablero, celdas, bolitas, colores, etc, en una versión proyectiva del lenguaje Java. Estas clases tienen el comportamiento específico del dominio y permiten

mantener el estado del programa en tiempo de interpretación. Las interpretaciones de los procedimientos y funciones nativos de Gobstones interactúan directamente con este modelo, de forma tal que, por ejemplo, la interpretación del procedimiento *Poner(<color>)* es un mensaje *addStones(Color color, int quantity)* que se le envía al tablero.

V-C. Edición del tablero inicial

Para editar el tablero inicial que constituye el input del programa gobstones, se crea un editor específico con un *layout table* como se observa en *Fig.7*. Este tablero se utiliza para inicializar la pila de tableros del estado del programa y puede definirse en un archivo virtual separado del del programa.

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.

Figura 7. Tablero inicial

V-D. 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 está construida con componentes pre-existentes en MPS, sino que se extendió el editor de MPS creando un componente de Swing[13] que toma un *Board* y lo renderiza como se aprecia en la *Fig.8*

Referencias

[1] 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.

[2] 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.

[3] 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.

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

[5] Markus Voelter y col. «Towards User-Friendly Projectional Editors». En: *7th International Conference on Software Language Engineering (SLE)*. 2014.

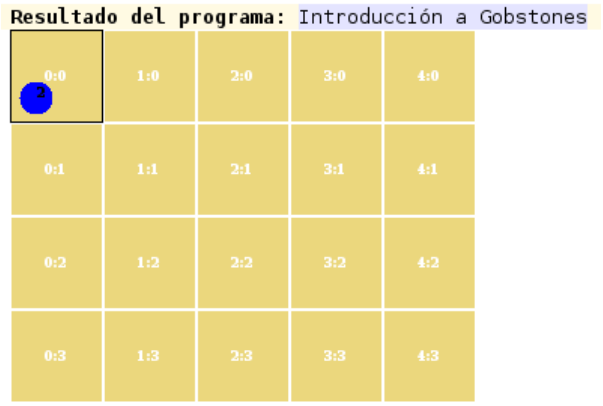


Figura 8. Tablero final

[6] Federico Sawady O'Connor Pablo Factorovich. *Actividades para aprender a Program.AR*. 2015. ISBN: 978-987-27416-1-7.

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

[8] Martin Fowler. *GUI Architectures*. URL: http://martinfowler.com/eaaDev/uiArchs.html.

[9] Lifelong Kindergarten Group. *Scratch*. URL: https://scratch.mit.edu/.

[10] itemis y fortiss. *mbeddr*. URL: http://mbeddr.com/.

[11] Campagne Laboratory. *MetaR*. URL: http://metaR.campagnelab.org.

[12] 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.

[13] Oracle. *Swing*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/swing/.

[14] JetBrains s.r.o. *MPW Workbench*. URL: https://www.jetbrains.com/mps/.

[15] Jetbrains s.r.o. *YouTrack*. URL: http://www.jetbrains.com/youtrack/.