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

ÍNDICE

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

***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. En el marco de una comunidad creciente de usuarios, tanto por su uso en cursos universitarios como por la adopción de la secuencia didáctica por parte de los cursos de Program.Ar[6], 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 facilitarle 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.**

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[13]; 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 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. Pretendiendo 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, resultó idóneo el uso de un *editor proyectivo*. Este término fue 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 VI sobre el sistema de chequeo de tipos, orientado a asistir al estudiante mediante mensajes de error legibles. En la sección VII 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 VIII 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 restriciones de features de lenguaje y análisis de código. Por último en la sección IX 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 decidió utilizar el workbench Meta Programming System (MPS)[15] 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[16]: 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 entorno de desarrollo autónomo o plugins para entornos pre-existentes.

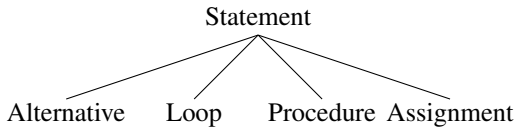
III. Modelo conceptual del lenguaje Gobstones

Se comenzó 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, 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.

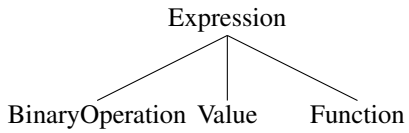
```
concept IfElseStatement extends     Statement
                        implements  <none>

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

  properties:
  << ... >>

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

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

Figura 1. Definición del concepto para la alternativa condicional

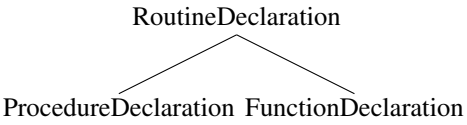
Los conceptos más importantes son *Statement*, que denota un comando, y *Expression* que denota una expresión que puede ser evaluada. De manera simplificada, tenemos que el primer nivel de la jerarquía de sentencias quedó dado por :



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



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

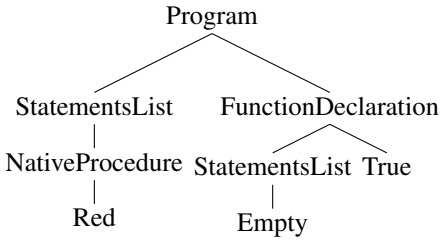


Decimos entonces que un programa gobstones básico, en el contexto de este editor proyectivo, se encuentra dado por una colección de sentencias y una colección de definición de rutinas.

Donde el siguiente programa:

```
program {
  Poner(Rojo)
}

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

Corresponde a un modelo conceptual con la estructura :



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 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.

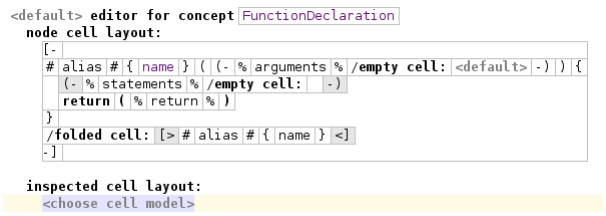


Figura 2. Definición del aspecto de edición para el concepto de declaración de función

En la *Fig.2* se observa el aspecto de edición definido para la declaración de funciones. En este quedan 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 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, al lograr dos vistas diferentes para un mismo modelo de tablero.

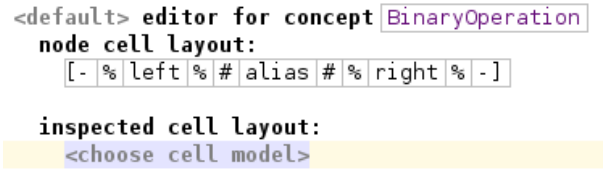
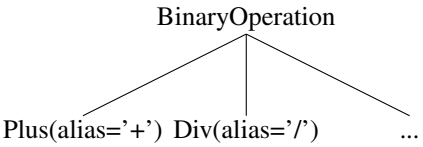


Figura 3. Definición del aspecto de edición para el concepto abstracto de operación binaria

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*[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 *Fig.4*.

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 4. Definición de estilos para la invocación de procedimientos

Se puede apreciar en *Fig.5* 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 el 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.
- **Autoidentación:** Las sentencias que se agregan son identadas 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.

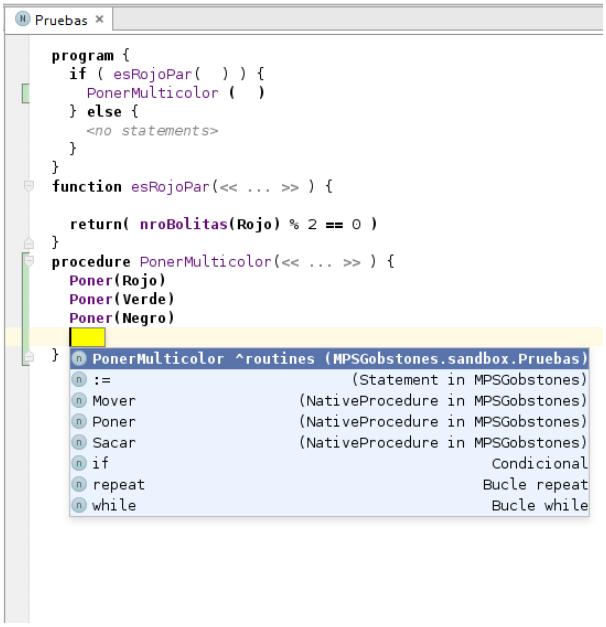


Figura 5. Editor proyectivo de Gobstones

V. 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 exclusívamente 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 a futuro será encapsular el comportamiento del intérprete en un aspecto distinto del de *behaviour*, y que podremos llamar *interpreter*.

Se definió entonces 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.6*.

```
concept behavior Statement {

  constructor {
    <no statements>
  }

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

}
```

Figura 6. 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.7* 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 7. 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 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 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 únicamente se evaluaría en caso de ser requerida. La complejidad extra de esta implementación radica en controlar los posibles *memory leaks* que se pudieran generar 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 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.8*, 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 8. 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 definieron 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 creó un editor específico con un *layout table* como se observa en *Fig.9*. 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.

nombre: un tablero inicial
filas: 5 **columnas:** 6

2 0	0 0	0 0	0 0	0 0	0 0
0 0	0 0	0 0	0 0	0 0	0 0
0 0	0 0	0 1	0 0	0 0	0 0
0 0	0 0	0 0	0 0	0 0	0 0
0 0	0 0	0 0	0 0	0 0	0 0
0 0	0 0	0 0	0 0	0 0	0 0
0 0	0 0	0 0	0 0	0 0	0 0
0 0	0 0	0 0	0 0	0 0	0 0
0 0	0 0	0 0	0 0	0 0	0 0
0 0	0 0	0 0	0 0	0 0	0 0

Figura 9. 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 fue construida con componentes pre-existentes en MPS, sino que se extendió el editor de MPS creando un componente de Swing[14] que toma un *Board* y lo renderiza, como se aprecia en la *Fig.10*. 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.

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 10. Tablero final

VI. 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 (UNQ 2012)*, 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.11*. En la *Fig.12* se ejemplifica un error durante el chequeo de tipos, el cual se muestra como un error legible en castellano.

```
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 11. Regla de inferencia de tipos para operaciones aritméticas

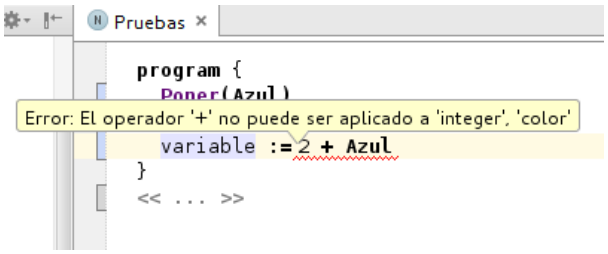


Figura 12. Ejemplo de alerta durante el chequeo de tipos

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.

VII. 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 isntancia 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.

Para llevar esto a cabo se declaran acciones de transformación y reemplazo de nodos, para las cuales se declaran condiciones disparadoras. Volviendo al ejemplo de las operaciones aritméticas (o, en general, de operadores binarios), se tiene que se comienza a escribir :

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.13*

```
program {
  variable := 1+
}
```

Figura 13. Edición de operaciones binarias sin acciones de reemplazo

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

De manera automática y sin que el usuario lo note, mostrándole *Fig.14*

```
program {
  variable := 1 + <número entero>
}
```

Figura 14. Edición de operaciones binarias con acciones de reemplazo definidas

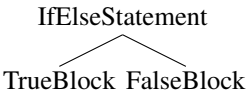
Algo similar ocurre con la alternativa condicional, pero esta vez cuando el usuario intenta borrar la rama falsa y dejar únicamente la verdadera. Al escribir el *if* se autogenera la plantilla correspondiente *Fig.15*

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

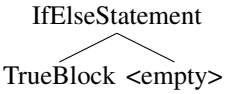
Figura 15. Editor de la alternativa condicional con dos ramas

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 esto, se volvió optativa la rama dalse, 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*. :



VIII. Definición de Ejercicios

Hasta el momento, las implementaciones existentes de gobstones carecían del concepto de proyecto como unidad de organización. En este trabajo se agregó esta idea 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.

Siguiendo la filosofía planteada en este trabajo, la definición del ejercicio también se visualiza como un archivo, orientado a ser legible y con apariencia de documentación *Fig.16*

Figura 16. Definición de un ejercicio simple

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 *Fig.17*

Figura 17. Definición de ejercicio con restricciones

Internamente, todo concepto del lenguaje que implemente la interfaz *CanBeRestricted* aparecerá en la lista de conceptos cuyo uso puede prohibirse.

IX. 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, pero por su misma naturaleza requirieron prestar particular atención a ciertos problemas de usabilidad que devienen de la no utilización de texto. La funcionalidad 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, no especificadas en *Las bases conceptuales de la Programación: Una nueva forma de aprender a programar* pero presentes en *PyGobstones*, 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 producto 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.

X. Anexo

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

XI. Agradecimientos

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

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.

[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] Pablo E. Martínez López. *Por qué gobstones es importante*. URL: http://gobstones-es-importante.blogspot.com.ar/.

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

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

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