El nombre verdadero de la programación

Una concepción de enseñanza de la programación para la sociedad de la información

Pablo E. Martínez López[*], Eduardo A. Bonelli[**], and Federico A. Sawady O'Connor[***]

Universidad Nacional de Quilmes

Cuando sepas reconocer la cuatrifolia en todos sus estados, raíz, hoja y flor, por la vista y el olfato, y la semilla, podrás aprender el nombre verdadero de la planta, ya que entonces conocerás su esencia, que es más que su utilidad.

Un Mago de Terramar
Úrsula K. Le Guin

Resumen En la sociedad actual, basada en la información y el conocimiento, se vuelve fundamental la manera en que se enseña programación. En este artículo explicitamos las bases conceptuales de una propuesta innovadora para un primer curso de programación, exhibiendo simultáneamente una metodología para la presentación de los conceptos deseados, y atendiendo la problemática de la formación previa de los estudiantes. Como parte de la tarea, presentamos una creación propia, el lenguaje Gobstones, diseñada para ser un vehículo que oriente el aprendizaje conciso de las herramientas abstractas de programación que deseamos enseñar.

1. Introducción

En el mundo actual, donde la innovación, y la creación, distribución y manipulación de información se ha vuelto un pilar para el desarrollo económico y cultural, especialmente de los países latinoamericanos, se vuelve fundamental la formación de los recursos humanos capacitados. En este marco, la necesidad de técnicos y graduados universitarios que posean adecuadas habilidades de programación es esencial, y por ello comprender las bases de la forma en que enseñamos

[*] fidel@unq.edu.ar
[**] ebonelli@unq.edu.ar
[***] sawady.faso@gmail.com

programación es una clave de este desarrollo. En la Argentina ha habido mucha actividad relacionada con el mejoramiento de la educación de programación, con proyectos de apoyo como el Fondo de Mejoramiento de la Enseñanza de la Informática (FOMENI)[1]o la revalorización de la educación secundaria con modalidad técnico-profesional[2]y ello propicia un aumento en el número de estudiantes que precisan formación específica en programación. Sin embargo, la base matemática y abstracta de este nuevo público para los docentes de programación es limitada o inadecuada, por lo que se impone estudiar la forma en que enseñamos a programar, y adecuar nuestros métodos.

Impartir un curso inicial de programación a personas que poseen una formación matemática incompleta o inadecuada, y que carecen de la capacidad de poder manipular abstracciones con soltura, se vuelve un desafío. Estas características cognitivas de los estudiantes afectan de manera directa el rendimiento académico y el proceso de enseñanza-aprendizaje en general, especialmente en disciplinas abstractas como la programación y la matemática. La enseñanza de la programación ha sido encarada desde diferentes ángulos a lo largo del tiempo. Generalmente, los enfoques tradicionales tienen como efecto secundario que muchos alumnos abandonen de forma temprana por falta de comprensión, aumentando notablemente los índices de desgranamiento en la Universidad, ya que dichos enfoques usualmente no consideran los conocimientos previos y el contexto anterior que los estudiantes poseen.

Uno de los caminos facilistas, muchas veces tomado para salvar dicha problemática, es simplemente intentar bajar el grado de dificultad de los conceptos que se van a enseñar, habitualmente utilizando diversas metáforas o analogías delineadas para tal fin, que pretenden resultar más amenas para el estudiante. Por otro lado, en numerosas ocasiones se comete el error de brindar demasiados conceptos en muy poco tiempo, sin focalizar aquellos que son fundamentales. Así pues, en muchos casos la elección y diseño de estos cursos iniciales se realiza de forma arriesgada, y no se contempla el objetivo de brindar una formación sólida al obviar o no presentar adecuadamente diversos conceptos que, a nuestro criterio, influyen sustancialmente en la formación de un buen programador, pero que a menudo se encuentran soslayados por sobreabundar detalles irrelevantes o complejos, cuando en el curso no se establecieron claramente las ideas que realmente funcionarán como pilares. Si bien muchos enfoques pueden parecer viables hasta cierto nivel, ciertas elecciones generan que luego al estudiante le resulte más difícil ser consciente de la existencia de ideas que le permitirían mejorar la producción y mantenimiento de su código, y que además le ayudarían a alcanzar y manipular, en un futuro, otras abstracciones avanzadas aún más complejas. Sólo unos pocos logran alcanzar un grado de conciencia notable a la hora de

[1] `http://cessiargentina.blogspot.com.ar/2007/09/`,
archivo: `el-fomeni-y-la-vigencia-de-un-plan.html`

[2] `http://infoleg.mecon.gov.ar/`,
carpeta: `infolegInternet/anexos/105000-109999/109525/norma.htm` y
`http://portal.educacion.gov.ar/`,
carpeta: `secundaria/modalidades/educacion-tecnico-profesional`

analizar qué ideas son importantes al momento de programar. El resto, que es mayoría, ve limitado su abanico de herramientas, lo que deteriora la calidad de la formación impartida, y en última instancia, la calidad del software que es producido.

En este artículo presentamos nuestra propuesta para encarar la enseñanza inicial de la programación atendiendo estos aspectos, así como las bases conceptuales de la misma, exponiendo además una herramienta que diseñamos para soportarla. Este enfoque se ha utilizado en los últimos 4 años en la Tecnicatura Universitaria en Programación Informática (TPI), carrera dictada en la Universidad Nacional de Quilmes (UNQ). Los alumnos de la carrera vienen logrando un buen aprendizaje de la programación, sin que esto represente un filtro excluyente de personas con ciertas falencias, tolerables al principio de su formación.

El objetivo de este artículo es presentar el enfoque utilizado, y explicitar el conjunto de conceptos que lo sustentan, porque creemos que bien impartidos son el pilar conceptual sobre el que otros conceptos más complejos pueden construirse con mayor facilidad. En la UNQ, previo al ingreso a la carrera existe un curso nivelatorio que brinda conocimientos básicos de matemática, física, química y producción de textos. Una vez que ingresan a la carrera, lo primero que cursan es Introducción a la Programación, materia tratada en este artículo. Materias siguientes tratan sobre estructuras de datos e introducción al paradigma de objetos. Nuestra idea es formarlos de tal manera que en las materias posteriores los alumnos manipulen cómodamente abstracciones básicas pero fundamentales en programación. Sin embargo, creemos que este enfoque no está limitado a la educación universitaria, sino que puede aplicarse también a la enseñanza de la programación en el nivel medio, puesto que no asume conocimientos específicos de matemáticas ni de lógica. De hecho, lo ideal sería comenzar a enseñar programación con un enfoque conceptual en el nivel medio, cuando los alumnos pueden captar todas las ideas de la mejor manera posible. La contribución más importante de este enfoque es que no depende de herramientas ad-hoc, que pueden ir mutando con el tiempo, sino en ideas sencillas, poderosas y perdurables. La dependencia de herramientas tecnológicas de moda, como programas de conexión virtual o simulaciones gráficas de ideas o asistentes electrónicos para sintaxis, no implica que la educación sea mejor, pues si bien facilitan la comprensión en el corto plazo, complican la posterior adquisición de elementos abstractos. En realidad, los enfoques sobre los que uno opera con los aparatos y herramientas también forman parte de lo que se denomina tecnología; una definición de *tecnología* es "el *conjunto de conocimientos técnicos*, ordenados científicamente, que permiten diseñar y crear bienes y servicios que facilitan la adaptación al medio ambiente y satisfacer tanto las necesidades esenciales como los deseos de las personas". En general se entiende por tecnología a las máquinas y aparatos en sí, a lo que ahora se suman los programas. Pero la tecnología trata sobre ideas, y por ende los enfoques y métodos de acercamiento a lo tecnológico también forman parte de ella. Esta es una de las razones por las que estamos convencidos del enfoque

que proponemos, que pretende formar programadores eficaces en relación a lo que la Sociedad de la Información exige.

Para focalizar adecuadamente los conceptos mencionados, diseñamos un lenguaje conciso fundamentado en éstos. El propósito es estimular y presentar convenientemente las abstracciones que deseamos impartir, minimizando detalles de ejecución y diversos elementos que consideramos no pertinentes. Sin embargo, nuestra meta final no es presentar un lenguaje de programación, sino atender en profunidad las ideas que presentaremos a lo largo del artículo, y que dan sustento a nuestra filosofía. Cualquier curso que elija como base nuestra metodología puede optar por tomar el lenguaje que construimos, o desarrollar uno propio que atienda igualmente aquellos aspectos sobre la enseñanza de la programación que estimamos importantes.

El artículo se organiza de la siguiente manera. Primero exponemos nuestra concepción acerca de programar y entender programas, aclarando qué clase de programador buscamos formar, qué conceptos creemos que son importantes al comienzo su formación y qué habilidades debe ser capaz de incorporar. Luego discutimos un modo de encarar el curso teniendo en cuenta el hecho de que las personas, lamentablemente, no vienen adecuadamente preparadas para ser formadas fácilmente de la manera en que pretendemos lo hagan. Posteriormente presentamos el lenguaje Gobstones y analizamos cada elemento que lo conforma, contrastando los mismos con los conceptos discutidos a lo largo del artículo. Por último, brindamos algunas conclusiones.

2. Concepción deseada de la enseñanza de la Programación

En esta sección, antes de presentar una selección propia de aquellos elementos que deseamos impartir en un primer curso, analizaremos cuál es la visión que da sustento a nuestra concepción de la programación, y qué aspectos debemos tener en cuenta para definir una formación inicial que sea independiente de cualquier paradigma, lenguaje o herramientas particulares.

2.1. Paradojas asociadas a la programación

> *La tarea de un pensador no consistía, para Shevek, en negar una realidad a expensas de otra, sino integrar y relacionar. No era una tarea fácil.*
>
> *Los Desposeídos*
> *Úrsula K. Le Guin*

Reconocemos dos paradojas relacionadas con la esencia de la programación, que guían nuestra concepción sobre su correcta enseñanza.

La primer paradoja está relacionada con el hecho de que es prioritario poder manipular ideas sin depender estrictamente de un lenguaje particular. Pero teniendo en cuenta que el lenguaje es el medio por el cual describimos estas ideas,

se vuelve a su vez imprescindible poder manipularlo con eficacia. Entonces, una forma en la que podemos enunciar esta paradoja es:

> "El lenguaje de programación que utilizamos no es importante, pero es extremadamente importante."

Lo que queremos indicar es que el lenguaje que utilizamos no es importante, porque debemos enfocarnos en las ideas a estudiar antes que en las características únicas de un lenguaje; pero a la vez es importante, porque es el único medio para poder describir, conceptualizar y comunicar ideas, por lo que debemos aprender a dominarlo adecuadamente. Cada tipo de lenguaje posee características propias, que encauzan la forma en que describimos ideas. Sumando esto a que en un mismo lenguaje pueden existir diversas formas de materializar una idea, es necesario concentrarse en la esencia de dicha idea para que el lenguaje no se vuelva un obstáculo que nos aleje del entendimiento de la misma.

De esta manera, nuestra meta es formar programadores que posean una base conceptual sólida orientada a dominar ideas. Si bien debemos empezar el curso presentando un lenguaje, el objetivo debe ser aprender ideas abstractas que trasciendan el lenguaje de turno. Con el fin de aprender a implementar algoritmos que den soluciones a especificaciones de problemas, pretendemos que un programador bien formado llegue al punto en el que mediante ideas trabajadas mentalmente, elija cómo representarlas en determinado lenguaje. Para esto el alumno debe adquirir un buen nivel de entendimiento y manipulación en abstracto de las ideas que incorpora, persiguiendo una comprensión analítica de la semántica del código que escribe.

En contraste, en muchos cursos iniciales se enfocan demasiado en los elementos específicos de un lenguaje particular, y no priorizan el entendimiento de las ideas subyacentes en las herramientas de programación que el lenguaje brinda, impidiendo luego poder ser generalizadas. En muchos casos es posible que estas habilidades sean adquiridas luego por otros medios, normalmente derivados de la práctica. Pero este tipo de enfoque en la enseñanza genera profesionales que se encuentran finalmente limitados por los elementos concretos que acostumbran utilizar en el lenguaje con el que elijen programar.

Además, debemos observar que la complejidad de los lenguajes de programación, y por lo tanto la de los componentes que conforman el software, van en aumento. Si nos quedamos solamente con las herramientas o la forma concreta con la que actualmente desarrollamos programas, nuestra formación se verá limitada a largo plazo. Por ende, creemos conveniente minimizar aspectos accidentales del software [3], es decir, *aquellos que son secundarios y que pueden ir variando a lo largo del tiempo sin producir cambios críticos en el razonamiento y producción del software*. Por esta razón, elegimos concentrarnos en enseñar los aspectos más esenciales: valoramos el aprendizaje de ideas por sobre el manejo de herramientas, y por ende el razonamiento abstracto por sobre el razonamiento concreto. Creemos que si un programador se enfoca debidamente en estos aspectos, tendrá la capacidad de construir ideas que permitan generalizar las herramientas que irá aprendiendo. Así estará mejor capacitado para enfrentar todo tipo de problemas de índole computacional, ya que creemos que ésta es

la forma adecuada de encararlos. Las herramientas pueden variar a lo largo del tiempo, y en cambio, las ideas que son inherentes a la esencia de los programas logran mantenerse.

Por otra parte, la concepción de programa ha ido variando a lo largo del tiempo, e incluso actualmente varía entre diferentes paradigmas. Es importante tener presente una definición precisa de programa que se complemente con nuestros objetivos, ya que la misma impacta directamente en la concepción sobre la formación que queremos brindar. Si bien la tendencia últimamente es pretender razonar los programas mediante un pensamiento de alto nivel[9,1], en muchos cursos de programación aún comienzan exhibiendo a los programas como una mera secuencia de instrucciones, atentando contra la formación del programador, ya que su pensamiento final muchas veces se ve limitado bajo este enfoque.

Podemos observar que los programas son entidades finalmente operacionales, desde el momento en que deben ejecutarse de manera eficiente para obtener las soluciones buscadas. Pero a su vez describen transformaciones de información, interacción con diferentes componentes y elementos abstractos como tipos de datos, que son los elementos en los que debe concentrarse el programador al momento de concebir el programa. De esta manera, podemos identificar dos aspectos que definen a los programas: el *aspecto denotacional* que comprende *qué describe o denota el programa*; y el *aspecto operacional* que comprende *los pasos que realiza el programa para cumplir una tarea*. Consecuentemente, definiremos a los programas como *descripciones ejecutables de soluciones a problemas* (de índole computacional)[3]. Esto refleja que si bien los programas son meras descripciones, siendo este el *aspecto denotacional* de los mismos, deben poder ejecutarse para dar lugar a soluciones. Esta última característica forma parte de su *aspecto operacional*, del que no podemos escapar totalmente, pese a querer abstraerlo mediante lenguajes de alto nivel y diversas ideas abstractas. Análogamente, en todo momento el programador puede optar por razonar a partir de alguna de estas dos posiciones: desde un pensamiento operacional, de bajo nivel, concreto, o por el contrario, desde un pensamiento denotacional, de alto nivel, abstracto.

La segunda paradoja que modela el enfoque filosófico que hemos elegido, radica puntualmente en que siendo los programas entidades fundamentalmente operacionales, debemos pensarlos, entenderlos y enseñarlos a partir de sus aspectos denotacionales, que nos permiten poder manipularlos y dar soluciones postergando en principio detalles de ejecución. Podemos entonces enunciarla como:

> "Debemos entender a los programas olvidando que son entidades operacionales, pero sin olvidar que son entidades operacionales."

Esta idea está basada en que muchos de los elementos en los que debemos concentrarnos al concebir las descripciones de soluciones son independientes del orden de ejecución o del modelo de memoria, por nombrar algunos elementos que caracterizan la forma en que dichas descripciones van a ser evaluadas. Bajo

[3] Cabe destacar que esta definición no está limitada a ningún paradigma.

este punto de vista, concebir a los programas como instrucciones dirigidas a la computadora termina siendo limitante, ya que aleja nuestra atención de los elementos denotacionales que componen la descripción que queremos desarrollar. Así, aunque sepamos que si una mera descripción cuando no es ejecutable no representa un programa, nosotros vamos a ignorar a conveniencia el hecho de que la misma se ejecuta para poder razonar y concentrarnos solamente en aquellos elementos abstractos que conforman dicha descripción. Igualmente, como todos los programas poseen características operacionales y denotacionales, creemos que un programador debe ser capaz de entender la correspondencia entre ambos aspectos y poder tomar decisiones que privilegien a uno u otro en cada momento, pero sin descuidar ninguno.

2.2. Selección de contenido para un primer curso

Hemos observado que la forma en que resolvemos las dos paradojas mencionadas repercute directamente en cómo razonamos la programación, y por lo tanto a los programas. Habiendo transmitido esta concepción de la programación, y siendo nuestra prioridad el desarrollo de un adecuado manejo abstracto y simbólico de las ideas, presentaremos como una elección posible para un primer curso, un conjunto relativamente pequeño de conceptos que estimamos fundamentales para conseguir este tipo de pensamiento. Estos conceptos se pueden agrupar en 3 categorías:

- ***Elementos del lenguaje***
  Valores y Acciones
  Los valores representan datos y las acciones representan efectos.
  Expresiones y Comandos
  Las expresiones son descripciones de valores y los comandos descripciones de acciones (y sus efectos).
  Funciones y Procedimientos
  Las funciones y los procedimientos son mecanismos para que el usuario pueda nombrar grupos de expresiones y comandos respectivamente. Consecuentemente, las funciones no poseen efectos y los procedimientos sí.
  Formas de combinación de elementos
  Secuencia (registros, bloques de comandos), alternativa (valores enumerados, alternativa condicional –*if-then-else*–, alternativa indexada –*case*–), repetición (listas, repetición condicional –*while*–, repetición indexada –*repeatWith*, *for*).
  Parámetros y mecanismos simples de pasaje de los mismos
  Mecanismo otorgado por lenguajes de programación para abstraer código similar.
  Estructuras de datos elementales
  Visión denotacional de mecanismos que agrupan información, tanto heterogénea como homogénea.
- ***Manejo del lenguaje***

Manejo de sintaxis dura
Reglas estrictas que poseen los lenguajes de programación, a las que se somete un programador al escribir código.

Buenas prácticas (cuestiones de estilo)
Como usar bien elementos tales como comentarios, indentación, nombres de identificadores, entre otros.

- ***Herramientas abstractas***

División en subproblemas
Partición de una tarea en fragmentos más fáciles de resolver para luego ser unidos en una solución más completa.

Parametrización
Abstracción que realiza el programador para expresar tareas similares en un mismo procedimiento o función.

Parcialidad y Precondiciones
Análisis de las causas en las que funciones y procedimientos pueden fallar y de los requisitos para evitar dichos errores.

Esquemas de programas sencillos (recorridos)
Abstracciones de algoritmos elementales, en particular del procesamiento de secuencias de elementos.

Puede observarse una dualidad entre algunos de los conceptos: acciones y valores, comandos y expresiones, procedimientos y funciones, entre otros. Dicha dualidad se centra, por un lado, en un aspecto concreto de los efectos en la ejecución del programa (acciones, comandos, procedimientos); y por otro lado, conceptos relacionados con la esencia abstracta de la información y del software (valores, expresiones, funciones). Es importante notar que hacemos distinción entre expresiones que denotan sólo valores, y comandos que denotan acciones (y por ende efectos). Del mismo modo discriminamos funciones puras y procedimientos, teniendo estos últimos efectos laterales mientras que las funciones no. Creemos que esta separación de ideas resulta convienente en los lenguajes, y sobre todo el primer lenguaje con que se topa un iniciado, ya que al presentarse en su forma más pura es posible delimitarlas fácilmente.

Ahora bien, los elementos del lenguaje que generan efectos pueden parecer, al menos desde una primera mirada ingenua, menos abstractos que los elementos que denotan valores. Al producir efectos, su repercusión en el mundo real es fácilmente observable desde algún punto de vista. De hecho, los estudiantes lo perciben de esta manera, y les resulta más fácil internalizar elementos que describen acciones por dicho motivo. Pero es fundamental que un programador logre entender que nociones de ambas partes son sustancialmente abstractas y denotan ideas en los programas. Por dicho motivo, nos interesa abordar cada concepto enfocándonos en sus aspectos denotacionales, es decir, qué denotan o describen. Diremos así que los comandos denotan acciones y sus efectos, y las expresiones denotan valores. Por el contrario, si inicialmente sólo presentáramos estas nociones a través de sus aspectos operacionales, esta situación sería

más díficil de comprender[4]. Con este enfoque, el programador podrá elegir en cualquier momento qué características distinguir de cada elemento, ya sea las denotacionales o las operacionales.

Otra herramienta poderosa, en la cual se hace mucho hincapié a lo largo de todo nuestro curso, es la división en subproblemas. Nombramos de esta forma a la metodología que divide un problema complejo en partes más sencillas y fáciles de encarar, para luego unir cada parte y brindar la solución completa. Cabe mencionar que en este punto no nos interesa la eficiencia, y siempre que algo se pueda resolver en una subtarea, inculcamos que debe hacerse de dicha manera. Si bien es normalmente complicado intentar dividir unas pocas líneas de código en diferentes tareas, creemos que no debe caerse en el error de definir un ejercicio por completo en un mismo procedimiento o función, si es que pueden distinguirse y separarse adecuadamente las subtareas que lo componen. La razón de esta elección es mayormente pedagógica, y creemos que si esto no se respeta desde el inicio se vuelve complicado después inculcar este concepto para programas mayores, resultando posteriormente en un proceso de aprendizaje más costoso. Elegimos entonces dividir en subproblemas en todos los ejercicios realizados a lo largo de este primer curso de programación, por más fáciles o pequeños que puedan resultar. Es interesante observar que esta idea es idéntica al concepto de delegación manejado en programación orientada a objetos, aunque lo nombramos diferente. Y aunque no usamos el nombre *delegación* para esta idea, los estudiantes no tienen problemas en identificar este concepto cuando lo encuentran en cursos posteriores con dicho nombre.

Por otra parte, una herramienta importante en los lenguajes es la parametrización, que nos permite capturar similitudes en el código abstrayendo diferencias. Es una herramienta básica, ya que permite lograr generalidad y disminuir la complejidad del código. Se relaciona de manera fundamental con la noción de división en subtareas, ya que permite expresar diversas subtareas diferentes con una única pieza de código, a través de la generalización. Esto disminuye la cantidad de subtareas necesarias y facilita la detección de abstracciones poderosas. Requiere de bastante nivel de abstracción por parte del estudiante, y por eso es difícil, aunque necesario, presentarlo en un primer curso. Sin embargo, creemos que no debe ofrecerse más que la forma más básica de mecanismo de pasaje de parámetros (por valor), para enfocarnos en el entendimiento de la idea sin ahondar en detalles que resultan espúreos en este nivel.

Para desalentar la búsqueda de errores de forma operacional (operación comúnmente conocida como *debugging*), optamos por transmitir algunas nociones sobre el análisis de la parcialidad de procedimientos y funciones, guiados por precondiciones que los satisfacen, aunque sin utilizar un lenguaje formal para su expresión. Esto posibilita el razonamiento de las condiciones bajo las cuales nuestras funciones y procedimientos van a ser válidos, es decir, sabemos que no van a tener problemas en su ejecución. Esta estrategia ha dado grandes resulta-

[4] En muchas ocasiones sucede que los alumnos erróneamente trasladan la noción de efecto (intentando resaltar únicamente su operacionalidad) a las expresiones y sus operaciones, perdiéndose así su manipulación desde su declaratividad.

dos, y muchos alumnos no se valen de una herramienta de debugging, ni siquiera en materias avanzadas de la carrera. Ciertamente consideramos más poderoso razonar acerca de qué describe denotacionalmente una parte del software, en lugar de buscar en el código, línea por línea, dónde existe un problema. No sólo se solucionan problemas más rápido y con menor esfuerzo, sino que también se asegura al mismo tiempo mayor robustez. Además, este tipo de análisis durante la codificación, nos ayuda a razonar más detenidamente sobre lo que escribimos, en lugar de ensayar recetas rápidas por prueba y error. Por otra parte, su presentación en el primer curso de programación, permite en la materia posterior de estructuras de datos, continuar esta idea bajo la noción de invariantes de representación, que se utilizan generalmente para guíar el modelado de las estructuras de datos (muchas veces garantizando propiedades que permiten obtener mayor eficiencia).

Una última herramienta abstracta que queremos distinguir es la idea de *recorrido*, término que hemos acuñado para este primer curso de programación. Está basada en ideas tales como el uso de invariantes de ciclo [11]; la idea de folds [7] (especialmente *foldr*), y generalizaciones asociadas [12]; y el enfoque de esquemas básicos de algoritmos [13]. Un recorrido trabaja sobre una secuencia de elementos; la noción de recorrido representa la preparación correcta de una repetición para el procesamiento de dichos elementos. Esta idea es presentada en su forma iterativa en el primer curso, aunque en las materias posteriores se presentan versiones recursivas (recursión estructural) y abstractas (iteradores, métodos de colecciones, entre otros). El programador al pensar un recorrido tiene en cuenta:

- Inicialización del recorrido: de qué forma se va a presentar un resultado, cuál es el primer paso de procesamiento que tendría que darse.
- Condición de corte: bajo qué condición se va a continuar el recorrido, cuándo va a finalizar.
- Procesamiento del elemento actual: qué se va a procesar o qué paso se va a realizar en cada iteración
- Pasar al siguiente: de qué manera se pasa al siguiente elemento a procesar, considerando la existencia o no de dicho elemento
- Finalización del recorrido: qué paso debería realizarse para finalizar correctamente y, en el caso de las funciones, retornar el resultado buscado

Ilustramos con un ejemplo en GOBSTONES:

```
procedure PintarTablero()
{
  // OBJETIVO: "Pinta" el tablero de rojo
  //           (colocando una bolita roja en cada celda)
  // PRECONDICION: ninguna, dado que es una operacion total
  // OBSERVACIONES: se estructura como recorrido sobre columnas
  IrALaPrimeraColumna()        // inicialización
  while(hayOtraColumna())      // condición de corte
  {
```

```
    ProcesarColumnaActual()    // procesamiento de un elemento
    PasarASiguienteColumna()   // paso al siguiente
  }
  ProcesarColumnaActual()      // finalización
}
```

Es importante notar en este ejemplo que la división en subtareas se consigue utilizando procedimientos y funciones de nombre específico, que deberán definirse. Esto potencia la idea de recorrido, logrando recorridos simples pero extremadamente poderosos. Además, cada línea tiene asociado un propósito denotacional relacionado con el problema y con la naturaleza del esquema de recorrido. Esto mismo puede lograrse sin utilizar procedimientos y funciones, aunque es más difícil para el novato detectar las subtareas asociadas.

Entonces, podemos observar que estas repeticiones requieren una correcta inicialización, una condición bajo la cual va a procesarse el siguiente elemento si lo hay, el procesamiento mismo, el pasaje al siguiente elemento, y los pasos que se requieran para finalizar. La mayoría de las tareas que nos enfrentamos día a día en programación poseen repeticiones de algún tipo, y por ende recorridos, por lo que es imprescindible que todo programador sea consciente de los elementos generales de los que se vale una repetición para poder brindar una solución correcta. Finalmente, estas nociones también sirven de guía para algoritmos más complejos (por ejemplo, se puede expresar el algoritmo para encontrar la salida de un laberinto con bifurcaciones usando un recorrido, sin recurrir a la noción de recursión ni de backtracking).

Como hemos explicado anteriormente, el lenguaje también es importante, y no sólo las ideas abstractas de las que nos valemos para solucionar problemas. Por ende, contemplamos acostumbrar a los alumnos a manejar un lenguaje con sintaxis dura, reglas estrictas, que guían en última instancia el programa que la computadora va a ejecutar, y que forman parte de la disciplina que es la programación. Añadimos también buenas prácticas como indentación, uso de nombres claros de identificadores y uso de comentarios adecuados para ilustrar el código. Sobre esto, podríamos decir que, desde el punto de vista de la ejecución, el nombre exacto de un parámetro, función o procedimiento es irrelevante, ya que lo que importa es que usemos siempre el mismo nombre; sin embargo, un nombre bien elegido contribuye a inducir una visión denotacional, abstracta, adecuada. Lo mismo sucede con los comentarios: son ignorados durante la ejecución, y por lo tanto no importa para ese aspecto que estén, o lo que digan, pero un comentario bien elegido facilita el reconocimiento de las ideas abstractas involucradas y evidencia que el programador las ha manejado. Finalmente lo mismo sucede con la indentación: (en la mayoría de los lenguajes) la indentación es irrelevante para la ejecución; sin embargo una buena indentación resalta la estructura del código y evidencia su comprensión. Tanto la elección de nombres como la de comentarios o la indentación son cuestiones de estilo, pues su uso depende mucho de qué ve cada programador cuando piensa su código, y en general no está relacionada con los aspectos operacionales. Creemos que es importante que los estudiantes

aprendan desde el comienzo a tomar gusto por el buen uso de estos elementos, evidenciando su pensamiento denotacional.

2.3. Discusión sobre contenido adicional

En esta subsección analizaremos algunos elementos más complejos o avanzados que los que hemos presentado, que podrían sumarse como contenido del curso o verse postergados para cursos subsiguientes por ser excesivos o no viables para un curso incial de programación. Particularmente nos explayaremos sobre el modelado de información, la noción de entrada/salida, el tratamiento de estructuras de datos, la introducción de conceptos relacionados a los paradigmas funcional y de objetos y la idea de realizar formalizaciones de los conceptos impartidos, argumentando las elecciones que hemos tomado en el diseño de nuestro curso.

El tema del modelado de información es muy importante en la programación. Sin embargo, en un primer curso no queremos enfocarnos en eso, pues requiere un nivel de abstracción importante que, como veremos, los estudiantes aún no poseen. Aunque utilizamos casi desde el comienzo modelados de información básica, y sobre el final del curso se explicitan algunas de estas ideas, no es nuestra intención profundizar este aspecto. El modelado de información se retoma como tema de estudio en las materias posteriores de estructuras de datos y programación con objetos. Puesto que debemos ser conscientes del volumen de contenido que administramos para el curso, no debemos caer en el error de tratar de enseñar más elementos de los que sería posible aprender correctamente. Pensando en un período de un semestre, consideramos que los elementos que explicamos anteriormente son los suficientes para comenzar con la formación de un programador de primer año, obteniendo buenos resultados en la formación del pensamiento abstracto.

Otro tema que es muy común agregar en el primer curso de programación, y que, de hecho, casi ni se cuestiona, es la noción de entrada/salida de información en los programas. Esta noción usualmente va asociada a la idea de ingresar datos por teclado y mostrarlos por pantalla, y suele ser parte inherente de los ejercicios de un primer curso de programación (e.g. ingresar una secuencia de números y mostrar su promedio, o leer ciertos datos de un archivo y escribirlos en otro archivo). Sin embargo, esta idea es fundamentalmente operacional, y conlleva en su misma concepción la idea de efecto y de modificación *in place* de ciertas estructuras, ambas nociones que superan la complejidad que creemos necesaria encarar en un primer curso. Por ello nosotros decidimos explícitamente dejar la interacción de entrada/salida delegada al mínimo imprescindible, que consiste exclusivamente en visualizar información que resulta de ejecutar un programa, pero sólo luego de su finalización. Ninguno de los ejercicios propuesto para el curso hace uso de nociones de entrada/salida, sino que todos se basan en la idea de parámetros.

Sobre estructuras de datos en un primer curso, es aconsejable que, de impartirse, sean las mínimas posibles e independientes de una memoria que las almacene, para poder entender sus operaciones y consecuente naturaleza, antes que

su implementación imperativa con un modelo de memoria dado. Para un primer curso destacamos la presentación de listas y no de arreglos por varias razones. Una razón es que deseamos acostumbrar al programador a resolver problemas de forma estructural, por lo que preferimos presentar primero estructuras de datos consistentes en tipos algebraicos[5] simples, aunque no las exhibamos en estos términos[6]. De hecho creemos que esta es la forma de comenzar a estudiar estructuras de datos. De esta manera, presentamos listas mediante sus constructores y selectores, sin ahondar en su implementación, pero que luego convenientemente aprovechamos para realizar recorridos sobre su estructura[7]. Otra razón radica en que los arreglos necesitan introducir la idea de índice para ser accedidos, es decir, no permiten tratar fácilmente la idea de recorrido estructural que deseamos transmitir inicialmente. Además, poseen un tamaño fijo que complejiza más de la cuenta ciertas situaciones, están más cerca de su implementación concreta, y la operación de asignación tiende a interpretarse de forma destructiva. Nuestro objetivo para un primer curso es presentar las primeras estructuras de datos desde una visión puramente denotacional, evitando así la formación de cualquier pensamiento de parte del estudiante que concierna con la destrucción o modificación *in place* de las estructuras. Por dicha razón elegimos presentar estas dos formas de combinación de elementos. Por un lado listas, que son una forma homogénea de combinación que representa la repetición de elementos, y por otro lado registros, que son una forma de combinación heterogénea que representa la secuenciación de elementos. En consecuencia, consideramos que ambas estructuras son básicas y se complementan precisamente con nuestros fines.

Una opción que exploramos en el primer dictado de la materia, y que abandonamos rápidamente, es la de utilizar el paradigma de programación funcional. El problema es que los lenguajes funcionales, y consecuentemente las ideas que se transmiten con ellos, son fundamentalmente abstractos. Además, puesto que en general se trata de lenguajes puros, recurren a herramientas complejas para reemplazar la noción de efecto, y esto no es fácilmente transmitible en un primer curso de las características que nos planteamos. Finalmente observamos que los elementos que decidimos impartir en nuestro curso son necesarios al aprender un lenguaje funcional, y por lo tanto decidimos postergar este paradigma para cursos posteriores.

Bajo esta misma línea, una cuestión que podemos analizar es si debemos impartir diseño de objetos en un primer curso de programación. El razonamiento basado en el paradigma de objetos es el más utilizado hoy en día, tanto en la industria como en la educación, ya que permite modelar programas operando con objetos que intentan proveer una forma natural de resolver problemas, per-

[5] Un tipo de dato algebraico es aquel que queda definido por sus constructores. Sobre este tipo podemos definir funciones inspectoras, que extraen información que dichos constructores poseen como argumento.

[6] Las nociones de Tipo Algebraico y Tipo Abstracto son vistas en profundidad en la materia de Estructuras de Datos.

[7] Las listas son un tipo recursivo simple de suma de datos, o sea diversas alternativas. Esto mismo es lo que nos ayuda a introducir recorridos estructurales sin mayor complejidad.

mitiendo manejar complejidad y haciendo a los programas altamente extensibles si se siguen buenas prácticas. No obstante, nuestra elección es postergar la programación orientada a objetos para un curso subsiguiente. Para el primer curso elegimos enseñar ideas como parametrización, división en subtareas, recorridos, manejo de sintaxis dura, funciones, procedimientos, etc. Si nos aventuramos al mismo tiempo en este paradigma, nos enfrentamos a muchas otras ideas más: objeto, mensaje, clase, polimorfismo, encapsulamiento, interfaz, etc. Consideramos que estos últimos elementos requieren el manejo de los elementos previos antes mencionados, que forman parte de cualquier lenguaje de programación, y no de un único paradigma. Es decir, en el primer curso buscamos prioritariamente formar un nivel de abstracción mayor antes que extendernos en detalles sobre otros temas sutilmente avanzados en abstracción de la programación (nos explayaremos sobre esto en la sección 3).

Para finalizar, deseamos que el alumno aprenda a ser preciso en la nomenclatura de los conceptos que forman parte de los contenidos, para que en posteriores materias esté capacitado para operar con estos términos de forma precisa. Aunque la formación a brindar sea más de tipo técnica que científica, los alumnos deben ser capaces de poder aplicar todas estas ideas, reconociendo dónde y por qué son necesarias, es decir, utilizándolas de manera pertinente y crítica, aunque su meta no sea teorizarlas. Un error común en muchos cursos es intentar ofrecer un nivel de formalización de estos conceptos en forma demasiado temprana. El problema es que la formalización requiere al mismo tiempo una comprensión de los conceptos a formalizar y de la herramienta de formalización, que, una vez más, es excesivamente abstracta. Por ello decidimos, nuevamente, soslayar o postergar (según se trate de una formación técnica o científica), esta noción.

Todo esto forma parte de un curso de introducción a la programación que creemos adecuado para que un programador sea suficientemente competente y comprenda cuáles son las herramientas básicas y esenciales que posee para programar a un nivel más abstracto, denotacional y de alto nivel. Podemos observar que estas nociones, además de trascender paradigmas de programación, deben entenderse a priori, antes de sumergirnos en un paradigma funcional, de objetos, estructurado, etc.

3. Problemática de querer impartir tal concepción

> *Programar es un desafío. La programación requiere el uso de habilidades de pensamiento abstracto y muchos estudiantes no han recibido asignaturas que requieran o les hayan ayudado a desarrollar estas habilidades en niveles precedentes de enseñanza, considerando que la asignatura de Programación debe ser una de las primeras que enfrente un estudiante de carreras de computación. Los estudiantes entran a la clase de programación con escasas habilidades conceptuales para programar.*
>
> *J. Díaz.[4]*

En esta sección discutiremos los problemas que dificultan el aprendizaje de la programación que pretendemos generar. No sólo existen inconvenientes provienentes de falencias de los estudiantes, sino también de la incorrecta presentación de los elementos analizados en la sección anterior. De esta manera, comenzaremos describiendo a grandes rasgos las características que observamos en la mayoría de los estudiantes y el contexto que encontramos en el aula actualmente. Posteriormente, explicaremos algunos errores que pueden cometerse al impartir la concepción de la programación antes presentada, analizando una posible metodología para alcanzar eficazmente nuestros objetivos. Consideramos así que no sólo son importantes las características técnicas aunque esenciales de la programación, sino también los fundamentos sobre la correcta adquisición y abordaje del conocimiento relacionado con esta disciplina y la computación en general.

3.1. Descripción del contexto

Es notable que la manipulación lingüística y el pensamiento matemático de la mayoría de los ingresantes a carreras de ciencias exactas y tecnología son altamente mecánicos y deficientes. Los alumnos evidencian un análisis pobre de los problemas que se les plantean y eso ciertamente va en detrimento de su pensamiento abstracto y manipulación simbólica a nivel general. Por esta razón, creemos conveniente no comenzar la enseñanza de la programación afrontando problemas clásicos de matemática (descomposición en factores, sistemas de numeración y operaciones sobre ellos, definición de la Sucesión de Fibonacci, etc.), ya que no podemos suponer que van a ser entendidos correctamente.

Por nuestra parte hemos observado algunas causas ligadas al bajo rendimiento académico de los alumnos:

- Concepción pobre de estrategias para abordar problemas de diversa índole.
- Falta de rigurosidad y costumbre a métodos formales de razonamiento.
- Dificultad en el entendimiento de metáforas y analogías.
- Bajo nivel de distinción de elementos esenciales por sobre los accidentales.
- Poco tiempo dedicado al estudio por tener que realizar otras tareas que consideran más prioritarias (trabajar, por ejemplo), o por mayor atención hacia otros elementos culturales.
- Escasez de personas capacitadas que cumplan el rol de orientadores en el proceso de aprendizaje (además de los profesores asignados a cada materia).

Nuestra tarea no es analizar en profundidad todos estos factores, pero sí planificar nuestra enseñanza teniendo en cuenta este contexto. Asímismo, creemos que la solución no es bajar el nivel de complejidad de los conceptos, ni de la calidad de educación impartida, sino encontrar una forma de organización de nuestra enseñanza que permita nivelar a los estudiantes con el fin de encarar las ideas que planteamos anteriormente.

3.2. Metodología elegida para afrontar esta situación

El tema en cuestión es: ¿Cómo explicar ideas abstractas de programación a personas totalmente desacostumbradas a entenderlas? Forzosamente debemos

partir desde un universo de discurso que contenga conceptos fácilmente razonables, pero que igualmente se correspondan con nuestros objetivos. Nuestra metodología se centra en un proceso de migración desde lo concreto hacia lo abstracto, puesto que nos vemos obligados a partir desde elementos concretos y cotidianos, por no requerir tanto esfuerzo para ser construidos mentalmente [8]. Pero la idea es migrar gradualmente a situaciones en donde lo que se manipula es inevitablemente reconocido como intangible, abstracto. En otras palabras, trabajamos con elementos aparentemente concretos, aunque más amenos, para conducir el desarrollo del pensamiento abstracto buscado.

En cierto modo, los elementos auxiliares de los que nos valemos para presentar las ideas principales también son abstractos, pero son presentados como concretos para los estudiantes, y éstos así lo creen en un principio. De esta manera es aconsejable que en determinado punto del curso se dediquen unos minutos a comprobar efectivamente que los elementos en los que se basaban los alumnos para codificar son representaciones completamente abstractas. Esto permite al sujeto registrar el proceso de abstracción que está realizando, reconociendo dichos elementos como meras construcciones mentales, para luego tener la capacidad de operar sólo con ideas abstractas y representaciones simbólicas, eliminando de manera consciente detalles superficiales según sea conveniente. Este es precisamente el tipo de pensamiento que buscamos en un programador, que elige convenientemente de qué manera modela la información en el lenguaje que desea utilizar, para que el programa describa la solución esperada.

No obstante, debemos analizar profundamente qué metáforas y analogías emplearemos para poder alcanzar este proceso de migración. En determinadas circunstancias las metáforas son necesarias, pero debemos utilizarlas de forma pertinente y crítica. Generalmente las comparaciones son punto de partida para el entendimiento de una idea, convirtiéndose usualmente en una salida fácil, pero el objetivo debe ser intentar transmitir de la forma más precisa posible la idea a comunicar originalmente, tratando de abarcar todas sus características. Debemos tener en cuenta que las propiedades de un objeto y la representación figurada de éste, pueden ser bien distintas. Las metáforas sirven y existen para ser usadas en situaciones específicas, ya que el contexto en el que son fundadas generalmente otorga validez a su significado. Por ende, extender la metáfora generalizándola o utilizándola como idea equivalente en cualquier contexto, puede provocar defectos en las conclusiones que se deriven de su uso. Mayor es el problema, si la metáfora se convierte exactamente en el instrumento cognoscitivo del que se vale el sujeto receptor para operar luego, utilizándola en razonamientos analógicos arriesgados, en vez de encarar los problemas entendiendo realmente las propiedades del objeto que dicha metáfora evoca.

Teniendo en cuenta dicha situación, también sería beneficioso entender que al momento de enseñar ideas abstractas en programación deben elegirse cuidadosamente los recursos gráficos a utilizar. Si las ideas abstractas son mayormente presentadas desde el inicio de manera gráfica, se limita a futuro su manipulación en abstracto, ya que el alumno puede optar por la representación gráfica y operar sólo con ésta [8], por estar más habituado a manipular ideas de esta forma o

por paracer más atractiva a los sentidos. Pero de esta manera se estaría dejando en segundo plano la idea abstracta *per se*, que es la más práctica y adecuada al momento de razonar. Asímismo, existen conceptos que son muy díficiles de representar de manera gráfica, o bien porque su representación de forma visual posee detalles demasiado distractorios y no pertinentes, o porque no refleja convenientemente muchas de sus características esenciales, que deben ser el foco de atención. En cualquier caso, si el estudiante no está acostumbrado a operar de forma abstracta, seguramente le será más difícil comprender algo de esta índole. En resumen, las herramientas gráficas típicamente resultan en una reducción del manejo del objeto de estudio, si justamente el concepto requiere operar con ideas abstractas e intangibles para su entendimiento. Además, esta situación es característica de la mayoría de los conceptos de las ciencias de la computación [5]. Por ende, entendemos que nuestra tarea debe ser motivar y guiar al alumno para que construya esquemas cognitivos que permitan una manipulación simbólica y abstracta, y consideramos que si incorpora dichas herramientas, va a contar con un incentivo adecuado para que pueda seguir aprendiendo ideas que no estén limitadas por la manera gráfica con la que se elige representarlas.

Del mismo modo, es aconsejable que de la ejecución de los programas se distinga únicamente el efecto final, o que al menos el alumno no intente representar cada paso intermedio desde la especificación a la solución. Esto es tentador, pero puede encauzarnos a utilizar un pensamiento primordialmente operacional. Si bien cada paso intermedio que realiza un programa puede ser importante, la descripción del programa en su totalidad debe ser el foco de atención para analizarlo y el punto de partida para corregir errores. Debemos mirar el programa desde una dimensión más amplia, y razonar así dónde puede encontrarse el error, y cómo podría corregirse. También esto ayuda a encontrar diferentes soluciones a un mismo problema.

En un primer curso de programación tampoco debería toparse el alumno con cuestiones de diseño. Es recomendable que todos los ejercicios y evaluaciones sean guiados, y apelen poco a cuestiones de estructuración del software. Por el contrario, debemos centrarnos en evaluar la habilidad del estudiante para expresar soluciones de forma clara y su capacidad de distinguir en el proceso de programación cada concepto que le es impartido (cómo divide en subtareas, elección de nombres de identificadores, uso de comentarios, etc).

Nuevamente podría considerarse introducir el paradigma de objetos, ya que hemos dicho que el contexto amerita empezar el curso razonando programas en base a elementos fácilmente manipulables. Igualmente volvemos a remarcar que nuestra meta es enseñar un grupo reducido pero sólido de conceptos que consideramos previos a muchos conceptos encontrados en el mundo de los objetos. Valernos de este paradigma puede ser tentador, pero consideramos que si el estudiante internaliza las ideas que le transmitimos en el primer curso de programación, la presentación de este paradigma en un curso posterior debería facilitarse enormemente. Incluso peor sería intentar introducir elementos de programación funcional al mismo tiempo que conceptos básicos de programación. Como hemos dicho, los alumnos ingresan con escasas habilidades matemáticas,

y esto haría extremadamente difícil poder abordar conceptos dentro del paradigma funcional por ser esencialmente abstractos. De hecho esto se intentó en nuestro curso la primera vez que se dictó y no tuvo buenos resultados.

Cabe señalar que las prácticas dentro del curso no sólo son llevadas a cabo en la computadora, sino que realizamos ejercicios en papel, que es el método elegido para los exámenes. Destacamos que esto nos ha dado buenos resultados, ya que para los iniciados es una buena forma de entrenar el volcado de las ideas que se van aprendiendo, maximizando su claridad, y experimentando además los errores cometidos en papel, que son fácilmente recordados y llegan a formar parte de un aprendizaje significativo. Del mismo modo, consideramos que el curso debe poseer una mirada positiva ante los errores cometidos durante el mismo. En nuestro caso tratamos de compartir los errores que va teniendo cada alumno, usualmente de forma anónima, para no intimidar al alumno que tuvo el error.

Por otra parte, los conceptos impartidos en ejercicios deben ser útiles y generalizables, acorde a la teoría vista en el curso, y no soluciones "ad-hoc" propias de muchos cursos iniciales, que por ejemplo, presentan como primeros ejercicios recorridos sobre arreglos u operaciones de entrada y salida, como tomar una cadena de caracteres o imprimir un "hello world". Estos ejercicios tradicionales no contribuyen a la formación de un estudiante capaz de discernir las abstracciones que debe emplear al programar y alientan un pensamiento operacional que limita a priori el razonamiento por subtareas y su posterior uso como "caja negra". Por el contrario, deben presentarse problemas que permitan identificar partes que representan tareas más simples o abstracciones que generalizan una situación dada, para luego unirlas y dar lugar a soluciones, siendo el objetivo pensar su utilización en base al qué resuelve y no el cómo.

Finalmente, un punto importante a analizar es el primer lenguaje de programación que se utilizará en el curso, ya que el lenguaje impacta notablemente en la dificultad o comodidad de un buen aprendizaje de la programación. Creemos que el primer lenguaje con el que se topa el alumno debe poseer sintaxis simple de razonar para un novato, y sobre todo, guiados por la metodología que elegimos perseguir, contener tipos básicos y operaciones primitivas altamente intuitivos. Desarrollamos Gobstones con este fin, por considerar pertinente proveernos a nosotros mismos de un lenguaje que posea exclusivamente los elementos que consideramos necesesarios.

4. El Lenguaje Gobstones

En esta sección analizaremos los elementos que posee el lenguaje que diseñamos, incluyendo la forma en que pretendemos sea utilizada, enfocándonos sólo en los aspectos que posee como herramienta pedagógica. Una descripción más detallada del lenguaje se encuentra en otro documento [10].

Fundamentalmente fueron dos las causas que nos llevaron a desarrollar nuestro propio lenguaje. La primera fue concebir un lenguaje que posea una clara separación entre los elementos que producen efectos (acciones, comandos, procedimientos), y los elementos puros (valores, expresiones, funciones). De modo

que, por un lado, nos permita hacer énfasis en entender el paradigma imperativo, pero sin descuidar al mismo tiempo la pureza de ciertos aspectos como las expresiones, basándonos en el paradigma funcional [6]. Nuestra segunda intención era eliminar elementos indeseados, como operaciones de entrada y salida, arreglos, manejo de memoria explícito, diferentes mecanismos de pasaje por parámetro, etc., que hubiésemos encontrado en otros lenguajes de uso general. Como remarcamos anteriormente, éstos desvían el foco de atención sobre las ideas que eficazmente intentamos transmitir.

El resultado fue GOBSTONES[8], un lenguaje conciso de sintaxis razonablemente simple, orientado a personas que no tienen conocimientos previos en programación. Sólo posee tipos básicos de datos, y no agrega nociones complejas de alcance de variables o elementos globales. Tampoco posee múltiples mecanismos de pasaje de parámetros (sólo pasaje por valor). El lenguaje actualmente no posee tipado estático, aunque sí validación por inferencia de tipos.

Adicionalmente, vimos la necesidad de idear un universo de discurso simple, para incluirlo en el lenguaje, con el fin aprender a resolver problemas en programación, pero al mismo tiempo intentando volver atractivo el aprendizaje, para lograr captar la atención y capacidad de asombro del estudiante. Este universo incluye un tablero con celdas, un cabezal que apunta a una celda del mismo, direcciones (norte, sur, este y oeste), números naturales con sus respectivas operaciones, y bolitas de colores (azul, rojo, negro y verde). El lenguaje provee comandos en forma de primitivas para mover el cabezal hacia una dirección, y también comandos para poner y sacar bolitas de un color en la celda a la que apunta dicho cabezal. Estas abstracciones (el tablero, el cabezal, las bolitas, los colores y las direcciones) son los elementos concretos de los que partimos para fomentar el desarrollo del pensamiento abstracto de los estudiantes. Si bien el tablero podría ser considerado similar a una memoria, ya que nos permite almacenar y leer información, este enfoque no es el que perseguimos y evitamos fomentar esta idea. Todas las abstracciones que forman parte del lenguaje son vistas de manera denotacional, remarcando solamente su utilidad para realizar descripciones que solucionan problemas utilizando como medio el tablero.

Bajo determinado punto de vista, podría argumentarse cierta similitud con otros lenguajes propuestos para empezar a enseñar programación (Logo [2] y otros similares), pero consideramos que nuestra aproximación es diferente. GOBSTONES fue diseñado con la pureza requerida para que podamos impartir nuestro enfoque de la programación alentando un pensamiento denotacional. Se vuelve difícil reproducir este enfoque si el lenguaje que nos orienta completamente a pensar de forma operacional. Por otra parte, si bien la meta es simplificar algunas cuestiones, GOBSTONES no esconde al programador la foma en que haría algoritmos que resuelvan problemas del mundo real. Esto significa que lo que se aprende no está ligado enteramente al universo de discurso del lenguaje, que solamente representa un medio para focalizar ideas. Además, la transición desde GOBSTONES a lenguajes usados en el mundo real es bastante simple, ya que su sintaxis fue pensada para ser similar a otros lenguajes imperativos convencionales (en

[8] Sitio SourceForge de GOBSTONES: *http://sourceforge.net/projects/gobstones/*

particular C y Java). Todas estas razones nos permiten refutar comparaciones con otros lenguajes inciales, que a nuestro criterio no contribuyen a la correcta formación de un programador.

Como hemos explicado, a pesar de que el lenguaje posee sintaxis imperativa, presentamos los conceptos pretendiendo generar un pensamiento denotacional. Por esa razón, elegimos entender a un programa GOBSTONES como *una descripción de las acciones que el cabezal intentaría realizar al ejecutar dicha descripción*. En el fondo, un programa está formado por comandos, descripciones de acciones individuales que generan efectos sobre el tablero. No existen otros efectos más que los generados por los comandos sobre el tablero[9], y existe una clara separación entre los elementos que generan efectos (acciones, comandos y procedimientos) y los elementos puros (valores, expresiones y funciones).

Siguiendo esta línea, para continuar desalentando el pensamiento operacional, la herramienta que lo implementa sólo permite ver el efecto final de ejecutar un programa, ya que como dijimos en las sección 3.2, si bien resulta tentador, limita a futuro el entendimiento de programas más complejos en donde razonar basándonos en su flujo deja de ser trivial. Entonces, dicha herramienta sólo muestra como la salida del programa, el tablero resultante de ejecutarlo, sin mostrar cómo se llena el tablero. De todas maneras otorgamos la posibilidad de proporcionar un tablero que sirva al programa de situación inicial, para que los alumnos puedan probar rápidamente sus programas en diferentes contextos. La meta es aprender a codificar bajo determinada especificación, presentando de forma gradual problemas básicos de programación.

En una segunda parte del curso, comenzamos un proceso de migración más profundo, para desprendernos del tablero, y comenzar a resolver problemas utilizando listas y registros. Mientras los primeros recorridos son realizados sobre las celdas del tablero, en la segunda parte se terminan realizando enteramente sobre listas. Los primeros podían ser concebidos de forma concreta, pero para los segundos esto se vuelve más difícil. La idea es dejar de pensar en efectos concretos realizados sobre el tablero, para empezar a reconocer la transformación de información que realizan los programas sobre los datos que manipulan. Este es el último paso que realizamos en el curso para sumergirnos enteramente en el mundo abstracto, y es donde los estudiantes acostumbran a ver que realmente están trabajando con meras abstracciones. También nosotros insistimos constantemente para que lo vean.

Por último, subrayamos que el mundo que propone GOBSTONES es sólo una de las tantas formas de encarar las ideas que creemos pertinentes. Cada curso es libre de encontrar la suya, basándose o no en nuestra herramienta. Recalcamos que el lenguaje es menos importante que todas las ideas que hemos explicado en este artículo, y deben ser ellas las que conduzcan los objetivos del curso.

[9] No hay memoria y aunque, como mencionamos, podría entenderse al tablero como una forma de memoria más concreta, elegimos soslayar este hecho.

5. Experiencia

En la Tecnicatura en Programación Informática hemos llevado a la práctica esta concepción de la programación durante los últimos 4 años (2007-2011). Consideramos que pese a las dificultades iniciales que caracterizan actualmente a la mayoría de los estudiantes, hemos cumplido con las expectativas, ya que en los exámenes que tomamos existe un alto nivel de aprobación, y el nivel de pensamiento abstracto se comprueba en cursos posteriores. Además, a los alumnos se les dificulta relativamente poco aprender nuevos conceptos en materias como Estructuras de Datos, Programación con Objetos I[10] o Programación Funcional.

Nuestro curso se da en un semestre, pero sería interesante ver su efectividad en un curso anual. Aún así, la elección del contenido que impartimos actualmente fue cuidadosamente planeada. Pueden agregarse temas, pero siempre teniendo en cuenta el volumen de información que pretendemos que los estudiantes puedan asimilar en tiempo y forma.

En la UNQ variamos de lenguaje entre una materia y otra (incluso hasta 2 lenguajes en una misma materia) y observamos que la curva de aprendizaje de los estudiantes es alta. Creemos que esto va de la mano de nuestra insistencia sobre la importancia de las ideas por encima del lenguaje a utilizar. Pero lo más destacable es que los alumnos logran concebir la programación de la forma en que pretendemos lo hagan, evidenciando un pensamiento claro, y priorizando ideas como precondiciones, invariantes de representación, esquemas de recorridos, transformación de datos, barreras de abstracción, por encima, p.ej. de una herramienta de debugging. Creemos que para que la mayoría de los alumnos alcance este tipo de pensamiento debe darse una adecuada introducción a la programación, o de lo contrario resultaría más difícil la formación de estas ideas.

No obstante, en muchas materias existe un grado notable de deserción. Observamos que esto es un fenómeno propio del nivel de estudios universitario, pero muchas veces también está especialmente relacionado con las carreras de informática y computación. Los alumnos ingresan con una noción desacertada de lo que significa una carrera de esta índole, y muchos abandonan por esa razón, cuando se encuentran con que no es lo que pretendían estudiar. Igualmente, muchos otros estudiantes que no sabían de qué trataba la carrera, siguieron con sus estudios porque finalmente les resultó atractiva. Nuevamente, si el curso inicial de programación es presentado de forma incorrecta, estos alumnos también pueden terminar abandonando.

Para finalizar, muchos alumnos avanzados se encuentran trabajando en la industria actualmente. Los mismos observan una gran diferencia entre el enfoque académico recibido y el utilizado en la industria. Sin embargo manifiestan que lo que aprendieron realmente les sirve todos los días, aún cuando eso no lo veían al inicio. Por todo esto, estamos convencidos que nuestro enfoque ha resultado en una experiencia altamente positiva.

[10] Materia de introducción al paradigma de objetos. Actualmente en la carrera existen otras dos materias Programación con Objetos II y Programación con Objetos III, que continúan presentando temas relacionados con el paradigma.

6. Conclusiones

En este artículo hemos presentado nuestra concepción acerca de programar y entender programas, así como la forma en que debemos exhibir los conceptos en un curso inicial. Hemos explicado aquellas bases conceptuales que sustentan nuestra filosofía, y que otorgan el foco necesario para poder concentrarnos en la esencia de las cosas que deseamos transmitir. Aclaramos que más que dedicarnos solamente a enseñar lenguajes particulares, nos concentramos en las ideas en sí que transcienden a los lenguajes y que son de gran valor a largo plazo. Además, alentamos un pensamiento denotacional (basándonos en lo que describen de los programas) en contraposición a un pensamiento operacional (qué pasos realiza el programa para cumplir una tarea). Esta es otra característica del pensamiento que intentamos formar en los alumnos, que nos ayuda a poder presentar en cursos posteriores ideas aún más abstractas que las presentadas en el primer curso de programación. Todas estas ideas son las que realmente hacen la diferencia a la hora de formar buenos programadores, capaces de emplear herramientas abstractas de diversa índole sin estar limitados por lenguajes o herramientas concretas que sólo facilitan la tarea de codificar.

Por su parte, explicamos que el contexto no nos favorece al principio del curso, pero que igualmente si utilizamos las herramientas adecuadas en aula, podemos contrarrestar esta situación. Es fácil desviarse de las metas elegidas, empleando estrategias erradas a la hora de presentar los conceptos, por lo que debemos tratar de mantener el foco necesario para poder transmitir las cosas íntegramente a través de su esencia. La meta es presentar adecuadamente conocimientos de programación pero teniendo en cuenta en todo momento las características que poseen los alumnos actualmente. De esta menera, recursos gráficos mal empleados, metáforas desacertadas y otros elementos similares que utilizamos en el aula para entablar analogías e intentar aprovechar los conomientos previos de los estudiantes, pueden perjudicarnos si no son antes analizados cuidadosamente. Si entendemos completamente el enfoque exhibido en el párrafo anterior, nuestra capacidad para poder elegir las herramientas adecuadas debería ser mayor.

Finalmente presentamos Gobstones el lenguaje que construimos para nuestro curso inicial de programación. Explicamos que posee exactamente los elementos que precisamos para poder enforcarnos debidamente en aquellos conceptos que deseamos enseñar, y que posee cierta pureza que permite identificar y trabajar fácilmente con dichos conceptos. A su vez elimina todos aquellos elementos que consideramos distractorios como entrada y salida, arreglos, diferentes mecanismos de pasaje de parámetros, etc. Por otra parte, el universo de discurso que posee el lenguaje nos ayuda a poder establecer problemas que no requieren demasiados conocimientos previos (de matemática por ejemplo), y que también resulta ameno a los estudiantes.

Concluimos el artículo afirmando que la verdadera cuestión no trata sobre hacer las cosas diferentes, *¡sino de concebirlas diferentes!*.

7. Agradecimientos

Agradecemos a Valeria De Cristófolo, Nicolás Passerini, Fernando Boucquez, Leonardo Volinier, Pablo Tobia, Federico Balaguer, Francisco Soulignac, Pablo Barenbaum, Guillermo Polito, Elías Filipponi y Federico Mieres, por haber formado parte del grupo docente de este primer curso de programación en estos 4 años. A todos los estudiantes que sufrieron nuestro aprendizaje, y aprendieron a programar con GOBSTONES. A Fernando Boucquez y Pablo Barenbaum por sus contribuciones a las herramientas que implementan GOBSTONES. A Gabriel Baum y Luis Sierra, porque muchas de las ideas que finalmente logramos plasmar en este documento y en nuestro curso, se gestaron a lo largo de los años a través de las discusiones que sostuvimos. Al proyecto FOMENI, y a quienes lo concibieron e implementaron, por permitirnos contar con la financiación necesaria para desarrollar la carrera donde estas ideas están floreciendo. Y finalmente a la UNQ por proveer un rico entorno de trabajo que nos permitió poner en práctica todas estas ideas y mejorarlas.

Referencias

1. H. Abelson, G.J. Sussman, J. Sussman, and A.J. Perlis. *Structure and interpretation of computer programs*. Mit Press, Cambridge, MA, 1996.
2. J.M. Arias and JE Bélanger. *Manual de programación en LOGO para la enseñanza básica*. Anaya Multimedia-Anaya Interactiva, 1988.
3. F.P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, 1987.
4. J. Díaz. Enseñando programación con C++: una propuesta didáctica. *Revista de Informática Educativa y Medios Audiovisuales*, 3(7):12–21, 2006.
5. E.W. Dijkstra et al. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.
6. J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98, 1989.
7. G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.
8. J.A. Kaminski, V.M. Sloutsky, and A.F. Heckler. Do children need concrete instantiations to learn an abstract concept. In *Proceedings of the 28th Annual Conference of the Cognitive Science Society*, pages 411–416, 2006.
9. A.C. Kay. The early history of Smalltalk. *ACM SIGPLAN Notices*, 28(3):69–95, March 1993.
10. Pablo E. Martínez López and Eduardo Bonelli. Introducción a la programación en Gobstones. Technical report, UNQ, 2008.
11. L. Mannila. Invariant based programming in education–an analysis of student difficulties. *Informatics in Education*, 9(1):115–132, 2010.
12. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.
13. P.C. Scholl and J.P. Peyrin. *Schémas algorithmiques fondamentaux: séquences et itération*. Université Joseph Fourier Institut Grenoblois d'études informatiques, 1988.