$\Delta\Delta$Lang

Implementación de un lenguaje Pascal-Like

Matías Federico Gobbi

Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba
9 de mayo de 2020

Resumen

Este trabajo consiste en el diseño e implementación de un intérprete para un lenguaje de programación estructurado basado en el lenguaje *Pascal*, orientado al aprendizaje de algoritmos y estructura de datos. El mismo es utilizado actualmente en una materia de la facultad, contando con una definición informal. Existe una sintaxis concreta relativamente consolidada aunque no especificada, y la semántica está definida de manera intuitiva. Siguiendo un modelo de desarrollo en cascada, analizamos la información disponible a partir del dictado de la materia obteniendo una definición formal de la sintaxis abstracta. Luego diseñamos una sucesión de chequeos estáticos, como el sistema de tipos. Finalmente definimos una semántica *small step* a partir de la cual implementamos un intérprete interactivo en el lenguaje *Haskell*.

Índice general

Capítulo 1

Introducción

En el siguiente trabajo, desarrollaremos un lenguaje de programación para la materia *Algoritmos y Estructura de Datos II*. Antes de comenzar propiamente con la definición formal e implementación del mismo, lo correcto es presentar los objetivos que nos hemos impuesto para la creación del lenguaje, y las motivaciones que nos han impulsado al desarrollo de este proyecto. Por lo tanto, la siguiente sección introducirá al lector los distintos aspectos que influyeron la formación del lenguaje, y justificaron la realización de este trabajo.

1.1. Motivación

En la materia *Algoritmos y Estructura de Datos II*, durante varios años, se ha utilizado un pseudocódigo para la enseñanza de los distintos conceptos que se estudian en la misma. Debido a esto, el lenguaje que diseñamos (basado en este pseudocódigo) tendrá un fin didáctico, y busca ser otra fuente de aprendizaje para auxiliar el dictado de la asignatura. Los ejes principales de la materia consisten en el análisis de algoritmos, la definición de tipos abstractos de datos, y la comprensión de diversas técnicas de programación.

El objetivo del pseudocódigo es poder introducir a los estudiantes a nuevos conceptos y fomentar buenas prácticas de programación. Se utiliza para describir de forma precisa principios operacionales de los distintos algoritmos estudiados en la materia. Típicamente, se omiten detalles esenciales para la implementación de los algoritmos para favorecer el entendimiento de los mismos. No existe ningún estándar para la sintaxis o semántica del pseudocódigo, por lo que un programa en este pseudo-lenguaje no es ejecutable en el sentido que no puede ser traducido a una serie de instrucciones máquina.

Nuestra meta final con este proyecto, es poder tomar la totalidad de los fragmentos de pseudocódigo que se encuentran dispersos en los diversos contenidos de la materia, y transformarlos en un lenguaje completamente implementado, con todo lo que esto implica. Obviamente, nuestro principal desafío para cumplir nuestra tarea, es poder resolver las distintas ambigüedades y la falta de especificación que el actual pseudocódigo presenta.

1.1.1. Objetivos de la Asignatura

Durante el desarrollo de la materia, se pretende que el alumno adquiera diversos conceptos relacionados con los distintos temas estudiados en la asignatura. Algunos de los mismos son listados a continuación:

- Capacidad para comprender y describir el problema que resuelve un algoritmo (el *qué*), y diferenciarlo de la manera en que lo resuelve (el *cómo*).

- Suficiencia para analizar algoritmos, compararlos según su eficiencia en tiempo de ejecución y en espacio de almacenamiento.

- Hábito de identificar abstracciones relevantes al abordar un problema computacional, y aptitud para la especificación e implementación de las mismas.

- Familiaridad con técnicas de diseño de algoritmos de uso frecuente, y comprensión de diversos algoritmos conocidos.

- Contacto con la programación (principalmente en el lenguaje *C*) de algoritmos y estructura de datos.

- Aptitud para la utilización de diversos niveles de abstracción y adaptación a distintos lenguajes de programación.

1.1.2. Programa de la Asignatura

El contenido de la materia se puede dividir en tres unidades. En cada una de estas, se introducen nuevos conceptos que luego se reflejan en diversos fragmentos de pseudocódigo empleados para facilitar la comprensión de los mismos. En el diseño del futuro lenguaje, se deberán tener en cuenta todas estas cuestiones para poder crear una herramienta útil para complementar la enseñanza de la asignatura.

Análisis de Algoritmos

La primer unidad de la materia, se basa en el análisis de algoritmos. Inicialmente, se estudian distintas maneras de ordenar arreglos utilizando diversas técnicas, como *ordenación por selección*, *ordenación por inserción*, *ordenación por intercalación*, *ordenación rápida*, entra otras. Con estos contenidos básicos presentes, se enseña al alumno a contar operaciones de un programa, introduciendo de esta forma los conceptos de *orden* y *jerarquía* sobre la complejidad de un algoritmo. Para terminar, se introducen las recurrencias *divide y vencerás*, y se presentan otros algoritmos como la *búsqueda lineal*, y la *búsqueda binaria*.

Estructura de Datos

La segunda parte de la materia, presenta la noción de estructuras de datos. Se describen a los *tipos concretos* como un concepto relativo a un lenguaje de programación, donde se estudian elementos como los arreglos, las listas, los registros, y los tipos enumerados. Mientras, los *tipos abstractos* se presentan como una idea asociada a un problema que se quiere resolver. Se describe la diferencia entre la *especificación*, y la *implementación* de los mismos, y se enseña la importancia de la elección adecuada para estos. Se examinan diseños distintos para varios de los diversos *TAD's*, como el *contador*, la *pila*, la *cola*, y el *árbol*, junto con la eficiencia en tiempo o espacio de sus distintas operaciones. Además, se introduce el concepto de *manejo dinámico de memoria* de un programa mediante el uso de punteros.

Algoritmos Avanzados

La última unidad, presenta distintas estrategias conocidas para la resolución de problemas algorítmicos. Se introduce el esquema general de los *algoritmos voraces*, y se enseñan diversos algoritmos que se basan en esta idea como el de *Dijkstra*, *Prim*, y *Kruskal*. También se introduce al concepto de *backtracking*, resolviendo los problemas de la *moneda*, y la *mochila*, entre otros. Luego, se ve *programación dinámica* donde se visitan problemas previos, además de ver nuevos conceptos como el algoritmo de *Floyd*. Un último tema que se enseña en la materia, es la recorrida de grafos, y las distintas variantes para realizar la misma.

1.2. Características del Lenguaje

Una vez detallados los fundamentos en los que se basa el lenguaje, es hora de describir sus características más importantes. ΔΔLang es una formalización del pseudocódigo utilizado en la materia *Algoritmos y Estructura de Datos II*, por lo que está diseñado para enseñar conceptos fundamentales de forma clara y natural. Es un lenguaje imperativo similar a *Pascal*. Este último fue diseñado por *Niklaus Wirth* cerca de 1970. Algunos elementos básicos que comparten son:

- Una sintaxis verbosa, pero fácil de leer.
- Un tipado fuerte para las expresiones.
- Un formato estructurado del código.

Al solo contar con una definición informal e incompleta del pseudocódigo, tuvimos que enfrentarnos a problemas de ambigüedad y falta de especificación para la creación del lenguaje. Debido a esto, la transición de uno a otro puede no ser inmediata. De todas formas, la esencia de ambos es la misma. La sintaxis del lenguaje es rigurosa, en comparación al código de la materia que era flexible en este aspecto. La semántica del primero está definida de forma precisa, a diferencia del pseudocódigo que solo se contaba con la intuición del lector para interpretar la misma.

1.2.1. Tipado

Como mencionamos previamente, ΔΔLang posee tipado fuerte. Esto significa que no se permiten violaciones de los tipos de datos, es decir, dado el valor de una variable de un tipo determinado, no se puede usar la misma como si fuera de otro tipo distinto al especificado en el programa. En una primera instancia, no hay ninguna especie de conversión, implícita o explícita, para los tipos de las expresiones del lenguaje.

En ΔΔLang se ofrecen una serie de tipos nativos un poco más limitada a la utilizada en el pseudocódigo de la materia. Los mismos, a su vez, se pueden dividir en tipos básicos y en tipos estructurados. Al mismo tiempo, existe la posibilidad de definir nuevos tipos de datos en el lenguaje; aspecto fundamental para el desarrollo de la segunda unidad de la asignatura.

Los tipos nativos básicos del lenguaje son los enteros, los reales, los booleanos y los caracteres. Para manipular valores de estos tipos, se ofrecen las operaciones aritméticas y lógicas típicas. A su vez, también se encuentran definidas las operaciones de igualdad y orden para estos valores, a pesar que su aplicación no se limita solo a los mismos.

Por otro lado, los tipos estructurados incorporados son los arreglos y los punteros. Los primeros tendrán un funcionamiento similar a los especificados en el lenguaje *C*. Además, existirá la posibilidad de definir arreglos multidimensionales, y arreglos cuyos tamaños serán variables. Los segundos, permitirán el manejo dinámico de la memoria de un programa y serán útiles para la creación de nuevos tipos de datos.

Finalmente, el usuario podrá crear sus propios tipos de datos. Hay tres posibilidades para la declaración de estos. Los tipos enumerados representarán una enumeración de un conjunto finito de valores. Los sinónimos serán un renombrado de un tipo ya existente. Y las tuplas permitirán la creación de estructuras con múltiples campos. Todos estos elementos serán fundamentales para el estudio de los *TAD's* en la materia.

Similar a *Haskell*, en el lenguaje existen ciertas clases predefinidas que caracterizan el comportamiento de los tipos que las implementan. Estas son **Eq**, **Ord**, e **Iter**. La primera, será implementada por todos los tipos que se pueden igualar. La segunda, es satisfecha por las categorías de elementos que son ordenables. Finalmente, la última clase indica si cierta estructura puede ser recorrida de forma iterativa. Una vez que el usuario declara un tipo, puede implementar todas las operaciones que una determinada clase requiera, para convertir a su nueva estructura de datos en una instancia de la misma.

1.2.2. Polimorfismo

El lenguaje permite la declaración de funciones y procedimientos con polimorfismo paramétrico. Esto significa que con una única definición, independiente de los tipos específicos de las entradas polimórficas, las funciones y procedimientos tendrán la capacidad de ser aplicables a argumentos con valores de distinto tipo. A su vez, esto introduce la capacidad de trabajar con variables de tipo en el programa, cuyos tipos concretos serán resueltos en tiempo de ejecución.

Otra posibilidad que permite el lenguaje, es agregar restricciones de clases que deberán ser satisfechas por las variables de tipo que los procedimientos y funciones introducen en su prototipo. Con este refinamiento, uno puede abstraerse de los tipos específicos en la implementación, para solo considerar las operaciones básicas que los mismos proporcionan. De esta forma, podemos limitarnos a trabajar con valores que ofrecen ciertas propiedades como la de igualdad, la de orden, y la de ser iterables.

El polimorfismo que admiten las funciones y procedimientos del lenguaje no se limita solo a tipos. También existe una especie de polimorfismo para los tamaños de arreglos que se introducen en la declaración de los anteriores. Con esta posibilidad, se pueden definir funciones o procedimientos que operan sobre arreglos, independientemente del tamaño de estos. Esto introduce la capacidad de trabajar con tamaños variables de arreglos, cuyo tamaño concreto será resuelto durante la ejecución del programa.

1.2.3. Recursión

La recursión es otro de los temas fundamentales en el dictado de la materia. En particular, para la parte de conteo de operaciones, donde se estudia como calcular el *orden* de un algoritmo. Dentro del cuerpo de una función o procedimiento, se puede realizar una llamada recursiva a si mismo para continuar con la ejecución del programa. En estas situaciones, el lenguaje no presenta ninguna particularidad relevante como para ser mencionada en esta sección.

En cambio, donde si haremos una salvedad, es en la declaración de tipos de datos. En el lenguaje, solo se permite una clase de recursión muy limitada para los mismos. Para crear un tipo recursivo, solo hay una posibilidad bastante restrictiva, y es en la definición de una tupla que posea un campo de tipo puntero. Esta especie de recursión es lo suficientemente expresiva como para permitir la implementación de listas enlazadas en el lenguaje, concepto fundamental en el programa de la asignatura.

1.2.4. Manejo de Memoria

Una característica muy importante del pseudocódigo, que se sigue manteniendo en el lenguaje, es el manejo dinámico de memoria. Durante la segunda y tercera parte de la materia, este es un concepto central que acompaña al desarrollo de la asignatura. Para la implementación de *TAD's*, resulta un tema recurrente que sirve para comparar distintos diseños en base a su claridad, portabilidad, y eficiencia.

Mediante el uso de punteros, y la llamada de los procedimientos especiales **alloc** y **free**, el usuario puede hacer un uso explícito sobre la memoria utilizada por el programa. Con algunos conceptos similares a *C*, uno puede reservar memoria que será accesible mediante el uso de punteros. Durante la ejecución del programa, se podrá manipular la memoria reservada y, cuando ya no sea necesaria, se podrá liberar la misma.

1.2.5. Encapsulamiento

Una última característica relevante que fue tomada de los contenidos de la materia y sentó las bases para el desarrollo del lenguaje, es el encapsulamiento. En el pseudocódigo esta particularidad muchas veces era omitida, debido que un programa en este pseudo-lenguaje solo se podía ejecutar en el *aire*. Debido a esto, se contaba con una intuición sobre que era parte de la especificación y que era parte de la implementación de un tipo de dato, de manera informal.

Cuando pasamos al lenguaje formal, hay una clara diferenciación entre el código que se utiliza para especificar un *TAD*, y el código empleado para su implementación. En $\Delta\Delta$Lang se busca tener una clara separación en módulos para los elementos definidos de un programa. De esta forma, podemos abstraernos de los detalles propios de la implementación de una estructura de datos, y basarnos solo en su especificación para resolver cierto problema algorítmico.

1.3. Desarrollo del Intérprete

Debido que el objetivo de este proyecto es la creación de un lenguaje de programación, nuestro ideal es que el producto final del trabajo sea la implementación de un intérprete para el mismo. El desarrollo de esta herramienta no es una actividad trivial, por el contrario, su avance requerirá de múltiples iteraciones y se dividirá en distintas etapas donde, a medida que se progrese en el proyecto, habrá una retroalimentación mutua entre las mismas. Con respecto a este trabajo de tesis en particular, realizaremos la implementación para la primera versión de la fase de *análisis* del intérprete. En la misma, luego de definir la sintaxis del lenguaje, desarrollaremos tanto el parser como los chequeos estáticos que conforman esta etapa.

A continuación, daremos un breve marco teórico sobre distintas cuestiones que consideramos importante mencionar sobre el diseño del intérprete en general. Para el mismo, nos basaremos en los primeros capítulos de la bibliografía de Aho, Sethi y Ullman 1988.

Figura 1.1: Un compilador

Un lenguaje de programación es una notación para describir computaciones a personas y a máquinas. Pero para que un programa sea ejecutable, antes debe ser traducido a un formato comprensible para una máquina. Los sistemas de software que se encargan de esta traducción son denominados *compiladores*. De forma simple, un compilador es un programa que puede leer

un programa especificado en un *lenguaje fuente* y traducirlo en un programa equivalente en un *lenguaje objetivo*. En la imagen 1.1 se puede observar un esquema simplificado de esta idea.

Un intérprete es otra clase común de procesador de lenguajes. En lugar de producir un programa objetivo como resultado de una traducción, un intérprete simula ejecutar directamente las operaciones especificadas en el programa fuente, en base a las entradas suministradas por el usuario y retornando las salidas producidas como resultado de la ejecución. En la figura 1.2 se puede apreciar la diferencia esencial entre ambas clases de procesadores de lenguajes.

Figura 1.2: Un intérprete

Si entramos un poco más en detalle, podemos observar que el proceso de transformación está compuesto por dos etapas: *análisis* y *síntesis*. En el caso de la primera etapa, su desarrollo puede realizarse de manera idéntica tanto para compiladores como para intérpretes. En cambio, la segunda presenta diferencias sustanciales entre ambos. Debido que nuestro objetivo es implementar un intérprete para el lenguaje, nos concentraremos solo en el estudio de este.

1.3.1. Análisis

La parte del análisis divide el programa fuente en distintas piezas e impone una estructura gramatical a las mismas. Luego, utiliza esta estructura para crear una representación intermedia del programa fuente. Si la etapa de análisis detecta que el programa presenta errores sintácticos o incoherencias semánticas, entonces deberá proveer mensajes informativos para que el usuario pueda aplicar las correcciones adecuadas. En nuestro caso, la representación intermedia que utilizaremos serán los *arboles de sintaxis abstracta*. La transformación del programa fuente a nuestra representación intermedia, no es trivial. Para facilitar la misma, comúnmente se divide esta tarea en varias fases.

Análisis Léxico

En esta etapa, también llamada *fase de escaneo*, se analiza la entrada carácter por carácter y se divide la misma en una serie de unidades elementales denominadas *componentes léxicos*. Por cada componente léxico, la fase de escaneo produce como salida un *token* que pertenece a cierta categoría gramatical y posee una cantidad determinada de atributos con información relevante para las siguientes fases de análisis. En esta etapa, además, se filtran elementos como los espacios en blanco y los comentarios.

Análisis Sintáctico

Esta es la fase que comúnmente se denomina *parser*. El parser utiliza los tokens obtenidos en la etapa previa, para crear una representación intermedia de la estructura gramatical del flujo total de tokens. Como mencionamos anteriormente, nosotros utilizaremos un *árbol de sintaxis abstracta* como representación. Las fases posteriores del intérprete emplearán esta estructura gramatical para continuar el análisis del programa fuente.

Análisis Semántico

La última etapa del análisis es la de *chequeos estáticos*. La misma se encarga de verificar si las restricciones semánticas impuestas en la definición del lenguaje son respetadas. Una parte importante de este análisis es el llamado chequeo de tipos (*typecheck*). Comúnmente, esta fase toma como entrada la representación intermedia obtenida en la etapa previa, y le agrega las anotaciones de tipos adecuadas, necesarias para continuar el análisis en etapas posteriores.

1.3.2. Síntesis

La parte de síntesis de un compilador es muy diferente a la de un intérprete. Para el primero, tenemos que construir el programa objetivo utilizando la representación intermedia, junto con toda la información adicional recopilada en las etapas previas. Habitualmente, esta fase se divide en otras dos partes, la *generación de código intermedio* y la *generación de código objeto*. En la *generación de código intermedio* se obtiene una representación independiente de la máquina, pero fácilmente traducible a lenguaje ensamblador. En cambio, la *generación de código objeto* es totalmente dependiente de la arquitectura concreta para la que se esté desarrollando el compilador. Además, durante estas fases comúnmente se aplica algún proceso de optimización sobre el código generado.

Del otro lado, como el objetivo de un intérprete difiere con el de un compilador, sus etapas de síntesis también lo hacen en la misma manera. Para obtener los resultados del programa, debemos partir del *árbol de sintaxis abstracta* obtenido luego de la fase de *análisis* del intérprete y, junto con los datos de entrada sustentados por el usuario, simular la ejecución del programa en base a las acciones especificadas en el código del mismo. De esta forma, una vez finalizada la ejecución, se obtienen las salidas del programa. En la imagen 1.3 se puede observar la estructura de nuestro intérprete.

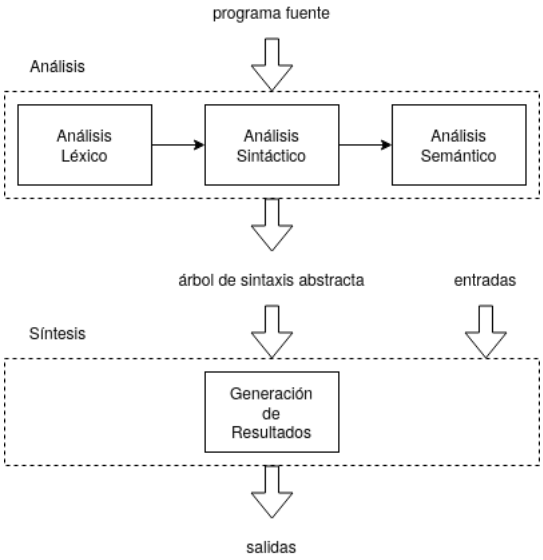

Figura 1.3: Estructura de un intérprete

Capítulo 2

Sobre el lenguaje

Capítulo 3

Sobre el parser

Capítulo 4

Sobre la semántica

(Reynolds 1998)

Bibliografía

Aho, Alfred, Ravi Sethi y Jeffrey Ullman (1988). *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

Reynolds, John Charles (1998). *Theories of Programming Languages*. Cambridge University Press.