UNIVERSIDADE DA CORUÑA

# Deseño das Linguaxes de Programación

## Práctica 1: Análisis de las características de un conjunto de lenguajes de programación a partir de un caso práctico


Autores

Rafael Alcalde Azpiazu
rafael.alcalde.azpiazu
(rafael.alcalde.azpiazu@udc.es)

Eva Suárez García
eva.suarez.garcia
(eva.suarez.garcia@udc.es)


Lenguajes analizados

Java
C
Python
OCaml

A Coruña - 23rd September 2017

# Contents

# 1 Implementación en Java

Para las pruebas en este lenguaje se usó el compilador `javac` en su versión 1.8.0_144 sobre Ubuntu Mate 16.04 (amd64).

## 1.1 Descripción

Java es un lenguaje de programación de alto nivel que sigue el paradigma de orientación a objetos basada en clases, aunque a día de hoy las últimas versiones ya incluyen elementos de otros paradigmas, como funciones-lambda.

Como en sus inicios se pensó en que fuera un lenguaje de propósito general, la compilación del código fuente se traduce a bytecode java, el cual es ejecutado sobre una máquina virtual que sí está diseñada para la plataforma final.

Su sintaxis deriva en gran medida de C++, siendo Java un lenguaje de tipado estático y fuerte. Es la propia máquina virtual la encargada de la gestión de memoria mediante un recolector automático de basura, el cual es responsable de gestionar el ciclo de vida, reservando memoria cuando el programador crea un objeto, y liberando memoria cuando se deja de referenciar completamente el objeto usado.

## 1.2 Análisis

### 1.2.1 Análisis de ventajas y desventajas

Como se ha mencionado anteriormente, Java no permite la gestión manual de la memoria, por lo que los tipos de datos que eran punteros no se han podido "traducir" directamente. La gestión de memoria automática nos permite olvidarnos de las tareas de reservar espacio, gestionar la falta de memoria (ya que la propia máquina virtual de Java se encarga de lanzar la excepción `OutOfMemoryError` cuando esto sucede) y liberar la memoria, posibilitando que nos centremos más en los algoritmos en sí.

Por otro lado nos encontramos con que en Java el paso de parámetros es siempre por valor y no por referencia, lo que ha ocasionado problemas que no existían en Pascal a la hora de manejar la estructura del árbol (explicaremos estos problemas más adelante cuando entremos en detalles de la implementación).

Por último, cabe destacar que al utilizar clases y no estructuras y punteros, el código resultante es de más alto nivel y más legible.

### 1.2.2 Análisis de implementación

En primer lugar hablaremos de la "traducción" de los tipos.

- `tPosA`, al ser un puntero, desaparece.

- `tNodoA` se convierte en la clase `Node`, con los mismos atributos que tenía como campos en la estructura original. Con el fin de ser fieles a la implementación en Pascal, todos los atributos son públicos.

- `tABB` se convierte en la clase `Bst`[1], el cual solo contiene un atributo de tipo `Node`. De este modo, Bst actúa como un envoltorio y punto de acceso a la estructura de nodos.

Volviendo sobre los problemas de paso de parámetros mencionados anteriormente, tenemos que, en concreto, cuando un objeto se pasa como argumento, lo que se recibe es una "copia" del puntero a dicho objeto, por lo que todos los cambios realizados en ese puntero no se reflejan fuera del método. Es por esto que ha resultado necesario recurrir a "punteros indirectos" como utilizar los nodos padre para manejar a los hijos, o utilizar el objeto Bst que envuelve la raíz para insertar y eliminar la misma.

Ahora hablemos de la adaptación de los procedimientos y funciones. Todos ellos se han convertido en métodos estáticos de una clase de utilidad. Los procedimientos y funciones que se mantenían como privados en Pascal siguen siendo privados en Java.

- `arbolVacio`: Crea un nuevo objeto Bst y lo devuelve. Difiere de la implementación en Pascal en que no inicializa un árbol pasado por parámetro, ya que si la inicialización se hiciera de esta forma no se reflejaría fuera del método por el paso de parámetros por valor de Java.

- `hijoIzquierdo`, `hijoDerecho` y `raiz`: Devuelven los atributos correspondientes del nodo alojado en la raíz del árbol. En el caso de los dos primeros, se devuelven envueltos en un nuevo objeto Bst para seguir la interfaz declarada en Pascal en cuanto al tipo devuelto.

- `esArbolVacio`: Se comprueba si el nodo alojado en la raíz es null.

- `insertarR`: En lugar de recorrer el árbol con el equivalente a un tABB, utilizamos un Node. Además, añadimos un nuevo argumento, de tipo Bst, representando el árbol cuya raíz es el nodo padre del Node. Debido al paso por valor de Java, el argumento Node no es más que una copia del puntero que queremos actualizar, por lo que no podemos hacer los cambios sobre este argumento. En su lugar, utilizamos el Bst padre, que sí contiene el puntero real que queremos actualizar accesible a través de su raíz. Inicialmente, Bst es el propio árbol en el que hay que insertar, lo que nos permite tener acceso real a la raíz en caso de que no exista.

- `insertarI`: De modo similar a la anterior, recibir el propio árbol nos permite insertar la raíz si esta no existiera, lo que resultaría imposible de recibir directamente el nodo raíz.

[1] Binary Search Tree

- `buscarR` y `buscarI`: Al igual que con `hijoIzquierdo` y `hijoDerecho`, el nodo a devolver se envuelve en un objeto Bst para cumplir con la interfaz de Pascal.

- `sup2`: La adaptación de este procedimiento recibe tres argumentos. Por un lado, el nodo que se va a eliminar para que podamos cambiar su clave por la mayor clave de su subárbol izquierdo. Por otro lado, el nodo que se utiliza para ir recorriendo el árbol hacia la derecha y el padre de dicho nodo. De esta forma, podemos mover los hijos izquierdos del nodo al nodo padre, algo que, de nuevo, no podríamos hacer si sólo tuviéramos el nodo en sí.

- `eliminarR`: Este método, además de la clave a eliminar, recibe un objeto Node que se usa para recorrer el árbol en busca del nodo a eliminar y un objeto Bst cuya raíz es el nodo padre, en la línea de insertR. Las diferencias entre la adaptación y el original se concentran en la eliminación de un nodo que tiene un hijo como mucho. Necesitamos añadir comprobaciones de si estamos eliminando la raíz del árbol completo, y en caso contrario si el nodo que estamos eliminando era un hijo izquierdo o derecho, ya que la reestructuración del árbol seguimos haciéndola desde el padre.

- `eliminarI`: Se adaptó de Pascal sin cambios relevantes.

### 1.2.3 Posibles mejoras

Al tener que respetar la implementación de Pascal, no han podido aprovecharse algunas de las ventajas de la orientación a objetos que habrían hecho la implementación en Java más natural y segura.

- Los atributos podrían haberse puesto como privados, implementando tan solo los getters y setters necesarios con la visibilidad apropiada para que el árbol no pudiera ser modificado desde el programa principal sin usar los métodos definidos para tal efecto. De esta forma, podría garantizarse que el árbol siempre mantuviera la estructura propia de un árbol binario de búsqueda.

- Los métodos podrían haberse implementado como propios de la clase Bst, en lugar de en una clase aparte. De este modo recibirían menos parámetros (ya que no necesitarían recibir el árbol a tratar) y la implementación sería más natural y propia de la orientación a objetos.

- Java nos permite definir las clases como paramétricas, lo que nos permitiría no tener que definir un árbol binario de búsqueda para cada tipo de clave, sino uno general en el que el tipo de clave se especificaría en el momento de declarar la variable Bst.

# 2 Implementación en OCaml

Para las pruebas en este lenguaje se usó el intérprete top-level `ocaml` en su versión 4.02.3 sobre Ubuntu Mate 16.04 (amd64).

## 2.1 Descripción

OCaml (Objetive Caml) es un lenguaje de programación de alto nivel que sigue principalmente el paradigma declarativo funcional, pero contiene métodos para la programación imperativa estructurada y orientación a objetos basado en clases. Basado en los Meta-Lenguajes, OCaml es la actual implementación de Caml (Categorical abstract machine language).

A pesar de que es un lenguaje de propósito general, el código fuente se puede compilar a bytecode para ser ejecutado sobre un intérprete top-level[2] que es el que está diseñado para la plataforma final, o mediante el compilador stand-alone[3] que proporciona código máquina para la plataforma actual.

Aunque es un lenguaje de tipado estático y fuerte, OCaml presenta un motor inferencial de tipos, pudiendo definir variables y tipos de datos polimórficos en tiempo de compilación. El compilador nativo y el intérprete top-level son los responsables de la gestión y asignación de memoria al compilar y en la ejecución de código OCaml.

## 2.2 Análisis

## 2.3 Modificaciones en el diseño

[2]El compilador es `ocamlc` y el intérprete top-level que ejecuta el código es `ocamlrun`.
[3]El compilador es `ocamlopt` y se ejecuta en la propia máquina.

# 3 Implementación en C

## 3.1 Descripción

## 3.2 Análisis

## 3.3 Modificaciones en el diseño

# 4 Implementación en Python

## 4.1 Descripción

## 4.2 Análisis

## 4.3 Modificaciones en el diseño