



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	2
1.1	Descripción	2
1.2	Análisis	2
1.2.1	Análisis de ventajas y desventajas	2
1.2.2	Análisis de implementación	2
2	Implementación en OCaml	5
2.1	Descripción	5
2.2	Análisis	5
2.3	Modificaciones en el diseño	5
3	Implementación en C	6
3.1	Descripción	6
3.2	Análisis	6
3.3	Modificaciones en el diseño	6
4	Implementación en Python	7
4.1	Descripción	7
4.2	Análisis	7
4.3	Modificaciones en el diseño	7

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. Para que se mantenga lo más parecido a Pascal, sus atributos son públicos y no existe ningún método de getter ni setter.
- **tABB** se convierte en la clase **Bst**¹, el cual solo contiene un atributo de tipo **Node**. De este modo

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 la estructura original. Siguiendo los principios de la orientación a objetos, los atributos son privados, pero tenemos getters públicos. Por otro lado, los setters tienen visibilidad paquete para que puedan ser usados desde la clase **Bst**, pero no desde el programa principal.
- **tABB** se convierte en la clase **Bst**², que tiene un atributo **Node** representando la raíz del árbol. De este modo, **Bst** actúa como un envoltorio y punto de acceso a la estructura de nodos. Todas las funciones y procedimientos definidos en el TAD de Pascal son métodos de esta clase.

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.

- **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**: Ya que para que el método pueda ser llamado el propio árbol no puede ser nulo, se comprueba si
- **insertarClave**:
- **buscarClave**: Al igual que con **hijoIzquierdo** y **hijoDerecho**, el nodo a devolver se envuelve en un objeto **Bst** para cumplir con la interfaz de Pascal.

¹Binary Search Tree

²Binary Search Tree

- **eliminarClave**: Tanto la adaptación de la versión recursiva de este procedimiento como del procedimiento auxiliar `sup2` reciben ahora un argumento más: el padre del nodo con mayor clave del subárbol izquierdo, necesario para poder reorganizar el árbol, ya que el paso por valor de Java no permite hacerlo de otra forma.

Además cabe destacar que las versiones iterativas de las funciones necesitan menos argumentos que su versión en Pascal, ya que no necesitan recibir el árbol a tratar.

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 (Objective 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-level³ que es el que está diseñado para la plataforma final, o mediante el compilador stand-alone⁴ 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. También es el propio compilador el responsable de la asignación de memoria en tiempo de compilado.

2.2 Análisis

2.3 Modificaciones en el diseño

³El compilador es `ocamlc` y el intérprete top-level que ejecuta el código es `ocamlrun`.

⁴El 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