UNIVERSIDADE DA CORUÑA

Programming Languages Design

Assignment 1: Programming languages analysis through a practice case

Authors

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

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

Analysed languages

Java
C
Python
OCaml

A Coruña - 29th 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
1.2.3 Posibles mejoras 4

2 Implementación en C **5**
2.1 Descripción 5
2.2 Análisis 5
2.3 Modificaciones en el diseño 5

3 Implementación en Python **6**
3.1 Descripción 6
3.2 Análisis 6
3.3 Modificaciones en el diseño 6

4 Implementación en OCaml **7**
4.1 Descripción 7
4.2 Análisis 7
4.3 Modificaciones en el diseño 7

1 Java's Implementation

Compiler used: `javac` version 1.8.0_144.
Operating System: Ubuntu Mate 16.04 (amd64).

1.1 Language Survey

Java is a general-purpose, high-level programming language that is object-oriented and class-based, although the lastest versions also include some elements from another paradigms, such as lambda-functions from functional programming. Much of its syntax derives from C++, but it has fewer low-level features.

When compiled, source code is translated to java bytecode, which is then executed via a virtual machine (VM) specifically designed for the final platform. Thanks to this, Java programs can run in a wide variety of platforms without having to recompile them. The VM itself takes care of dynamic memory management through an automatic garbage collector, which is responsible of allocating memory when an object is created and freeing it when said object becomes unreferenced.

Among other characteristics, we can mention that Java's typing is strong and static, and that all method arguments are passed by value.

1.2 Analysis

1.2.1 Advantages and Disadvantages

As stated before, Java does not allow manual memory management, so pointer data types could not be "translated" directly. On the other hand, automatic memory management allows us to focus on the algorithms themselves, forgetting about handling lack of memory (as Java's VM will throw an `OutOfMemoryError` in such a case).

Another important issue is the argument-passing method (ni idea de si lo que acabo de poner tiene sentido lmao). As parameters are always passed by value, handling the tree structure became more complicated (we shall explain this matter in more depth in the implementation details section).

Finally, as we use classes instead of structs and pointers, the resulting code is higher-level and easier to read.

1.2.2 Implementation

In first place, we shall talk about the "translation" of data types.

- `tPosA`, as it is a pointer, disappears.
- `tNodoA` becomes the class `Node`, which has the same attributes the original struct had as fields. With the goal of being true (ni idea de si está bien) to the Pascal implementation, all attributes are public.

- `tABB` becomes the class `Bst`[1], which only contains a `Node` attribute representing the tree root. In this way `Bst` acts as a wrapper for the structure of nodes and provides an access point to it.

Returning to the value-passed arguments problems mentioned earlier, we have the following situation: whenever an object is passed as an argument, what the method actually receives is a copy of the pointer to that object, so all changes done to the received pointer (such as assigning it to a new object) will not remain upon the method's end. This forces us to use some sort of "indirect pointers", that is, we use an object which is not the one we want to modify, but contains a reference to it. For example, to delete a node, we use its parent node.

Now, let us move on to the adaptation of procedures and functions. All of them have become static methods from a utility class. The ones that were private in Pascal remain private in Java.

- `arbolVacio`: Creates a new `Bst` object and returns it. Differs from the original implementation in that no `Bst` is passed as argument. Should it be done that way (nunca usé esta expresión así que igual está mal 3:), the `Bst` would remain unitialized (palabra sacada de la nada (?)) due to the passed-by-value arguments in Java.

- `hijoIzquierdo`, `hijoDerecho` and `raiz`: They return the pertinent attributes from the root node of the tree. For the first two ones, the nodes are returned wrapped in a `Bst` object to follow Pascal specifications with respect to the returned type.

- `esArbolVacio`: Checks if the root node inside the `Bst` is `null`.

- `insertarR`: Instead of traversing the tree with a `Bst` (the equivalent to the `tABB`), we use a `Node`. Besides, we add a new argument `Bst` representing the tree whose root node is the parent node of our `Node` argument. Remember that in this point, our `Node` argument is just a copy of the pointer we want to modify. By using the parent node, we can access the pointer we actually want to change. Initially, `Bst` is the tree we want to insert into, so that we have real access to the root node (and not a copy) in the case we need to insert in there.

- `insertarI`: In a similar way to the mentioned above, receiving the tree itself allows to insert the root if it did not exist.

- `buscarR` y `buscarI`: The found node is wrapped in a `Bst` object in order to follow Pascal's interface.

- `sup2`: This procedure's adaptation receives 3 parameters.

  - *Node to be deleted*, so that we can replace its key with the greatest one from its left subtree.

[1]Binary Search Tree

  - *Node* used as an iterator to traverse the tree to the right.
  - *Parent node* of the iterator node. This way we can move the iterator's left subtree to its parent.

- `eliminarR`: This method also receives 3 arguments.
  - *Key* to delete.
  - *Node* to use as an iterator to search for the node to remove.
  - *Bst* whose root is the parent node of the iterator node.

  M

- `eliminarI`: Adapted from Pascal's implementation without relevant changes.

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

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 también se puede utilizar el compilador stand-alone[3] que proporciona código máquina para la plataforma final.

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, es decir, no existe gestión manual de la memoria.

2.2 Análisis

2.2.1 Análisis de ventajas y desventajas

Al igual que en Java, no disponemos de gestión de memoria manual, con las implicaciones ya mencionadas.

El modo en el que el árbol está implementado en Pascal no favorece el uso de la programación funcional de una forma tal que las versiones iterativas y recursivas sean intercambiables, por lo que fue necesario apartarse un tanto de este tipo de programación.

Por otro lado, el sistema de matching que ofrece OCaml facilita la implementación de la versión recursiva de las operaciones.

2.2.2 Análisis de implementación

Nuestro árbol se representa mediante el tipo `bst`, que puede tomar los valores `Empty` y `Node(int, bst ref, bst ref)`. Utilizar `bst ref` como hijos derecho e izquierdo nos permite manejarlos de una manera similar a los punteros de Pascal.

[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.

Una `ref` es una estructura con un campo `contents`, que es la información que realmente se modifica (una especie de equivalente al contenido del puntero).

Con respecto a las funciones, tenemos lo siguiente:

- `arbolVacio:` Recibe `unit`[4] y devuelve una ref a Empty.
- `hijoIzquierdo, hijoDerecho` y `raiz:` Matchean el contenido del árbol con un Node(key,left,right) y devuelven el componente que corresponda. Al igual que en la implementación en Pascal, fallan si el árbol de entrada está vacío.
- `esArbolVacio:` Si el contenido del árbol es Empty devuelve true, en cualquier otro caso devuelve false.
- `insertar_r:` La traducción es directa teniendo en cuenta el sistema de matching de OCaml y el cambio de contenido de los refs.
- `insertar_i:` La traducción también es directa.
- `buscar_r:` La traducción es directa, pasando por un primer match propio de OCaml para ahorrar if-then-else.
- `buscar_i:` La adaptación es directa.
- `sup2:` Recibe el nodo utilizado para recorrer el árbol en busca del nodo con mayor clave, y el nodo padre de este. De este modo, una vez encontrado el nodo con mayor clave, usamos dicha clave para sustituir a la que se va a eliminar, y mediante el nodo padre recolocamos sus hijos izquierdos en el árbol resultado.
- `eliminar_r:` Se devuelve un unit si el árbol de entrada está vacío (es decir, si la clave no está en el árbol). En caso contrario, la adaptación desde Pascal se realiza sin cambios relevantes, a excepción de la función auxiliar `sup2` ya comentada.
- `delete_i:` La adaptación es casi directa, salvo por la introducción de una nueva variable auxiliar: parentMaxLeftChild, que representa al nodo padre de maxLeftChild y cumple con las funciones que correspondían a parentRm en Pascal. Por su parte, parentRm se reserva para cambiar la clave del nodo a eliminar desde ahí, ya que rm es una "copia" del nodo que realmente queremos modificar. Si parentRm es Empty (es decir, se va a eliminar la raíz del árbol) modificamos directamente a través del árbol pasado por parámetro.

2.2.3 Posibles mejoras

Al igual que en Java, en OCaml podría definirse un árbol binario de búsqueda para un tipo de clave genérico (especificado como 'a en la definición del tipo).

[4] El equivalente al vacío en OCaml

Por otro lado, el tener que mantenerse fiel a la implementación en Pascal no nos ha permitido utilizar el paradigma funcional propiamente dicho. De haberlo hecho de esta forma, habríamos tenido una implementación recursiva más intuitiva para el lenguaje y más sencilla al no tener que manejar el equivalente a punteros, si bien la iterativa resultaría imposible de realizar.

3 Implementación en C

Para las pruebas en este lenguaje se usó el compilador `gcc` en su versión 5.4.0 sobre Ubuntu Mate 16.04 (amd64).

3.1 Descripción

C es un lenguaje de programación de propósito general que sigue el paradigma imperativo procedural estructurado. Es un lenguaje de nivel medio ya que permite el uso de procedimientos y estructuras de alto nivel así como incluir código ensamblador en el código fuente.

La compilación del código fuente produce código máquina para la plataforma final. Como su principal uso fue dirigido a la creación de sistemas operativos, a día de hoy es un lenguaje multiplataforma. Sus compiladores permiten compilarlo para cualquier plataforma final sin hacer apenas cambios en el código.

Su sintaxis viene especificada en el estándar C, definiéndolo como lenguaje de tipado estático, débil, nominal y declarativo (manifest). La gestión de la memoria es manual, es decir, es el programador el encargado de reservar memoria dinámica y liberarla.

La enorme popularidad del lenguaje le ha permitido ser el que más implementaciones tiene. A día de hoy sirve de lenguaje intermedio para otros lenguajes de alto nivel como C++, C#, Objetive-C, Go, Perl, PHP, Python, Lua y Swift. También contiene muchísimas bibliotecas externas para emular comportamientos de otros lenguajes, como la gestión de memoria automática, orientación a objetos mediante GObject e incluso poder usar funciones lamda.

3.2 Análisis

C y Pascal comparten el mismo paradigma y la misma forma de trabajo, por lo que la adaptación a C ha resultado en un código casi idéntico al proporcionado en la implementación en Pascal. Sólo nos queda comentar un pequeños detalles que han marcado la diferencia entre los dos lenguajes.

El primero es que C no presenta un método estructurado en el propio lenguaje para definir un módulo. La solución más parecida es incluir en un archivo `.h` aparte las definiciones de tipos de variables y funciones que puede usar el programa principal, y en un archivo `.c` la implementación tanto de las funciones visibles al programa principal como de las funciones internas del módulo (como las formas recursivas e iterativas de inserción, borrado y búsqueda, y la función para crear un nuevo nodo e imprimir un error cuando no hay espacio). Esto no evita que el programador pueda hacer implementaciones en el archivo de cabecera, cosa que Pascal restringe más mediante la definición de módulos.

Otro problema que nos encontramos es que C no tiene una forma de indicar si los argumentos de las funciones son pasados por valor o por referencia tan sencilla y cómoda de utilizar como el VAR de Pascal. En su lugar, el paso por referencia se realiza pasando un puntero al parámetro (*********los parámetros como punteros), lo cual resulta un engorro de cara a la programación y la legibilidad del código.

Por último, para ser fieles a la implementación de Pascal, hemos tenido que usar la biblioteca `stdbol` debido a que C no contiene el tipo primitivo `boolean`.

Con esto podemos concluir que Pascal es más estructurado y de mayor nivel de abstracción que C, por lo que realmente no hay mejoras que C pueda hacer con respecto a Pascal en este caso.

4 Implementación en Python

4.1 Descripción

4.2 Análisis

4.3 Modificaciones en el diseño