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 - 30th September 2017

Contents

1 Java's Implementation **2**
1.1 Language Survey . . . 2
1.2 Analysis . . . 2
1.2.1 Advantages and Disadvantages . . . 2
1.2.2 Implementation . . . 2
1.2.3 Possible Enhancements . . . 4

2 OCaml's Implementation **5**
2.1 Language Survey . . . 5
2.2 Analysis . . . 5
2.2.1 Advantages and Disadvantages . . . 5
2.2.2 Implementation . . . 5
2.2.3 Possible Enhancements . . . 6

3 Implementación en C **7**
3.1 Descripción . . . 7
3.2 Análisis . . . 7

4 Implementación en Python **9**
4.1 Descripción . . . 9
4.2 Análisis . . . 9
4.3 Modificaciones en el diseño . . . 9

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.

[1] Binary Search Tree

- `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.
  - *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.

  More changes appear in the removal of a node with at most one child, as we need to check if this node is the root of the whole tree, and in other case if the node is a left or a right child, because we keep making the reorganisation through the parent.
- `eliminarI`: Adapted from Pascal's implementation without relevant changes.

1.2.3 Possible Enhancements

Due to the requirement to respect the implementation defined in Pascal, we coul not take advantage of some features of object-orientation that would have made Java's implementation of the tree more natural and safe.

- The attributes could have been private, implementing only the getter and setter methods needed with the apropiate visibility so that the tree could not be modified from the main program without using the method provided to that effect. This way we could guarantee that the tree always preserved the structure of a binary search tree.
- Methods could have been owned by the Bst class itself. This woul imply that methods would not need to receive the tree as an argument, and the resulting code would be more natural for an object-oriented language.
- Java allows us to define parametric classes, which would result in defining just one Bst tha would be valid for all types of keys (or at least, all key types with a comparation method). The key type would then be determined at the moment of declaring each Bst variable.

2 OCaml's Implementation

For this language we used the top-level interpreter `ocaml` version 4.02.3. Operating System: Ubuntu Mate 16.04 (amd64).

2.1 Language Survey

OCaml (Objective Caml) is a general-purpose high-level programming language that follows mainly the functional paradigm, but it has imperative and object-oriented facilities too. It is based off ML[2], OCaml is the current implementation of Caml (Categorical abstract machine language).

The source code can be compiled to bytecode to be then executed in/on a top-level interpreter designed for the final platform[3]. Another option is to use a stand-alone compiler`ocamlopt`, which provides machine code for the final platform.

OCaml's typing is static, strong and inferred. This language has a motor for inferring types, allowing to define polimorphic variables and data types.

The native compiler and the top-level interpreter are responsible for managing the dynamic memory during compilation and execution, that is, there is no manual memory management.

2.2 Analysis

2.2.1 Advantages and Disadvantages

As OCaml does not have manual memory management, we have the same implications as in Java in this aspect.

The way the tree is implemented in Pascal does not favor the use of functional programming in such a way that iterative and recursive implementations can be used interchangeably, so it was necessary to stay away from this kind of programming. (no sabía cómo decir lo que tenía en castellano y se nota xd)

Another thing to note is that the matching system OCaml provides makes easier implementing the recursive version of the operations.

2.2.2 Implementation

The tree is represented by the type `bst`. This type can have two different values: `Empty` and `Node(int, bst ref, bst ref)`. Using `bst ref` as left and right children allows us to manipulate them in a way similar to Pascal's pointers. A `ref` is a structure containing a field `contents`, which is the information that is

[2]Meta Language

[3]The compiler is `ocamlc` and the top-level interpreter is `ocamlrun`. The two of them can be used together with `ocaml` orders

actually modified (equivalent to the pointer's content in some way).
With regard to the functions, we have the following:

- `arbolVacio:` Receives `unit`[4] and returns a `ref` to `Empty`.
- `hijoIzquierdo, hijoDerecho` and `raiz:` They match the tree's content with a Node(key,left,right) and return the pertinent component. Like in Pascal's implementation, they fail if the tree is empty.
- `esArbolVacio:` Returns true if the tree's content is `Empty`, and false otherwise.
- `insertar_r:` The adaptation is direct taking into account OCaml's matching system and refs' change of content.
- `insertar_i:` The adaptation is direct.
- `buscar_r:` The adaptation is direct, usign a first match to avoid an if-then-else.
- `buscar_i:` The adaptation is direct.
- `sup2:` Receives the node used to traverse the tree while searching for the node with the greatest key, and the parent node to that one. In this way, once the node with the greatest key is found, we use that key to replace the one to delete, and reorganise its left child via its parent node.
- `eliminar_r:` A `unit` is returned if the received tree is empty (that is, if the key is not in the initial tree). Otherwise, the adaptation from Pascal is done without relevant changes, apart from those mentioned in the auxiliar function.
- `delete_i:` The adaptation is almost direct, but we introduce a new auxiliar variable: `parentMaxLeftChild`, which represents `maxLeftChild`'s parent node. It fulfills the role of `parentRm` in Pascal, only in the section where maxLeftChild is used, obviously. `parentRm` is used to change the key of the node to delete from there, as `rm` is just a "copy" of the node we actually want to modify. If `parentRm` is `Empty` (that is, the node to be removed is the root of the whole tree) we modify it directly from the tree passed as argument to the function.

2.2.3 Possible Enhancements

Like in Java, in OCaml we could have defined a binary search tree for a generic key type (specified as 'a in the type definition).

On the other hand, because it is required to stick to Pascal's implementation, we could not use the functional paradigm. Should we have done that, we would have a more intuitive recursive implementation given the language. It would have been simpler as well, as we would not have needed to use refs. However, the iterative version would have been impossible to implement this way.

[4]The equivalent to void in OCaml

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