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 - 6th October 2017

Contents

1 General Information

Operating System: All languages were tested on a laptop running Ubuntu Mate 16.04 (amd64).

Compilation: All implementations (except Python's) come with a Makefile to make compilation and execution easier. In the case of Python, a README with instructions to run the example program is provided.

Iterative/Recursive versions: In order to change between iterative and recursive implementations of functions, just navigate to the wrapper function (for example, `searchKey`), comment the current version and uncomment the other one.

Example programs: If you want to change the example programs, please remember that most Pascal's functions fail whenever an uninitialized tree is passed as an argument, so the other implementations will fail as well. Make sure you always use a properly initialized tree when you are testing the behaviour of the other functions.

2 Java

Compiler used: `javac` version 1.8.0_144.

2.1 Language Survey

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

When compiled, the 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 be run in a wide variety of platforms without having to recompile them. The VM itself takes care of the dynamic memory management through an automatic garbage collector, which is responsible for 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.

2.2 Analysis

2.2.1 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 parameter passing method. 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.

2.2.2 Implementation

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

- `tPosA`, as it is a pointer, it 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 when the method ends. 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 the utility class `BstUtils`. 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 an argument. Should it be done that way, the `Bst` would remain uninitialized 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 methods, the nodes are returned wrapped in a `Bst` object to follow Pascal's 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 `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 in which we want to insert, 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 inserting 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 specifications. In the first method, a `Node` argument is used instead of a `Bst` argument just for simplifying the code.
- `sup2`: This procedure 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 once we have finished traversing the tree to the right.

  As Java does not allow nested methods, it is implemented as a private method from the utility class.

- `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.

  Apart from the ones mentioned in `sup2`, 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 another case, if the node is a left or a right child, as we keep making the reorganization through the parent.

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

All of these classes belong to the package `bst`, while the main program lies inside the package `bstProgram`.

2.2.3 Possible Enhancements

Due to the requirement to respect the implementation defined in Pascal, we could 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 appropiate 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 would 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 that would be valid for all types of keys (or at least, all key types with a comparison method). The key type would then be determined at the moment of declaring each Bst variable.

3 OCaml

Compiler used: `ocamlc` version 4.02.3, together with the bytecode interpreter `ocamlrun`.

3.1 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. Being a member of the ML (Meta Language) family, OCaml is the current implementation of Caml (Categorical abstract machine language).

The source code can be compiled to bytecode which is then executed in an interpreter[2]. Another option is to use the native code compiler `ocamlopt`, which produces machine code for the final platform. Interactive use of OCaml is allowed through the toplevel system `ocaml`, which repeatedly reads OCaml phrases from the input, then typechecks, compiles and evaluates them, then prints the inferred type and result value, if any.

OCaml's typing is static, strong and inferred, and allows to define polymorphic 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.

3.2 Analysis

3.2.1 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.

Another thing to mention is that the matching system OCaml provides simplifies implementation as we use it to replace the first if-then-else statement, thus allowing us to obtain the components of a `Node` without using the other functions, which ultimately increases the readability of the source code.

3.2.2 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 chil-

[2]The compiler is `ocamlc` and the interpreter is `ocamlrun`

dren allow us to manipulate them in a somewhat similar way to Pascal's pointers. A `ref` is a structure containing a field `contents`, which is the information that is actually modified (equivalent to the pointer's content in some way).
With regard to the functions, we have the following:

- `arbolVacio:` Receives `unit`[3] and returns a `ref` containing `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, using a first match to avoid an if-then-else.
- `insertar_i:` The adaptation is direct.
- `buscar_r:` The adaptation is direct, using a first match to avoid an if-then-else.
- `buscar_i:` The adaptation is direct.
- `sup2:` It is implemented as a local function. 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 we are going to delete, and then reorganize 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 auxiliary function.
- `delete_i:` The adaptation is almost direct, but we introduce a new auxiliary 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 an argument to the function.

Note: Whenever an iterator variable is needed (that is, a variable for traversing the tree) it is created as a `ref` to the tree argument's content in order to have a copy of the node, thus preventing any undesired changes to the tree structure during traversal.

[3]The equivalent to void in OCaml

3.2.3 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 had a more intuitive recursive implementation given the language. It would have been simpler and easier to implement as well, as we would not have needed to use refs. However, the iterative version would have been impossible to implement this way.

4 C

Compiler used: `gcc` version 5.4.0.

4.1 Description

C is a general-purpose programming language that follows the *imperative*, *procedural*, *structured/*
paradigm. It is a medium-level language that allows the use of procedures and high-level structures as well as the inclusion of assembly code in the source code. Its initial main use was the creation of Operating Systems.

Compilation of source code produces machine code for the final platform. It is a cross-platform language, so its compilers allow compilation to a lot of final platforms making few or no changes *in/to* the source code.

The C standard defines C's typing as static, weak, manifest and nominal. The memory management is manual, that is, the programmer is responsible for reserving and freeing dynamic memory.

C's popularity has resulted in it having the greatest amount of implementations among programming languages. Nowadays, it serves as an intermediate language for other high-level languages such as C++, C#, Objective-C, Go, Perl, PHP, Python, Lua and Swift. It also has many external libraries to emulate features from other languages, such as automatic memory management, object-orientation through GObject or even lambda-functions.

4.2 Analysis

C and Pascal share the same paradigm *andworkmethod*, so the adapted code is almost identical to that of the implementation in Pascal. The only thing left is to mention some details that supposed the diference between both languages.

In the first place, C does not have a *really* structured way *todefine/ofdefining* a module. The most similar solution is to include in a `.h` file the definitions of variable types and functions the main program can use and in a `.c` file the implementation of both the public and the private functions (like the recursive and iterative versions of insertion, deletion and search, as well as the function to create a new node and print an error when there is no space left). This does not prevent the programmer from implementing functions in the header file, something that Pascal restricts more by its module definition.

AnotherissueisthatCdoesnothaveawaytoindicateifargumentsarepassedbyvalueorbyreferencea
Instead of this keyword, arguments passed by reference need to be passed as a pointer, which is annoying *withrespecttothetaskof//for* programming and reading the code.

Finally, as we needed to make it similar to Pascal's code, we used the library `stdbool` in order to have the `boolean` type, as C does not have it by default (it uses 0 as false and any other number as true instead).

We can conclude that Pascal is more structured and has a higher abstraction level than C, and as C's low-level features do not suppose any improvements, there are actually no enhances C could make with respect to Pascal in this case.

5 Python

Interpreter used: `python3` version 3.5.2.

5.1 Language Survey

Python es un lenguaje de programación de propósito general multiparadigma. Es un lenguaje de alto nivel e interpretado, que nació con la filosofía de enfatizar la legibilidad del código y una sintaxis sencilla y compacta para que los desarrolladores programaran con la menor cantidad de lineas de código posible en comparación a otros lenguajes como C++ o Java.

Python usa un tipado fuerte y dinámico y no declarativo, permitiendo duck typing. Contiene un recolector de memoria automática, siendo el intérprete el encargado de reservar y eliminar la memoria según procesa el código.

La implementación de referencia del intérprete de Python es CPython, el cual es open source y es administrado por la Python Software Foundation.

5.2 Analysis

5.2.1 Advantages and Disadvantages

La primera diferencia que nos encontramos en Python con respecto a Pascal es que Python es interpretado, permitiendo no tener que compilar el código cada vez que se modifica. Con respecto a esto podemos afirmar que el propio intérprete nos presenta más información cuando se produce un error que en el resto de lenguajes analizados, muy a la par del intérprete bytecode de Java.

Otra cosa que podemos comentar es que en Python no tenemos gestión manual de memoria, por lo que tenemos las mismas implicaciones de Java y OCaml que comentar en este aspecto.

Por otro lado, los parámetros de las funciones son pasados por referencia en Python. Esto nos ha permitido simplificar la forma de trabajar con el árbol, pareciéndose a la implementación de Pascal en muchos casos.

Un hecho en contra a comentar de este lenguaje es que al no existir método alguno de declarar el tipo de las variables, no hay forma de forzar al programador a que pase unos tipos de variables concretos a la función, dando error en ejecución. En nuestro caso, esto también implica en que el programador puede intentar insertar claves de distinto tipo.

Como ventaja de esto último, debido al carecer de forma de declarar los tipos, la propia implementación permite usar como clave cualquier tipo que permita usar expresiones de comparación.

Otro hecho en contra es que el propio lenguaje no tiene forma de definir variables o funciones locales / privadas a menos que sean dentro de diferentes scopes (variable definidas dentro de una función). Esto supone que todas las funciones definidas en el módulo principal son públicas a quien lo use, sin restricción alguna.

5.2.2 Implementation

La principal diferencia en la implementación con respecto a todos los demás lenguajes analizados ha sido la traducción de tipos. En Python las variables son de tipo dinámico y no se declaran, por lo que se ha eliminado toda la definición de tipos presente en los demás lenguajes.

Debido a esto, hemos definido el árbol como un diccionario (un array asociativo) que contiene tres claves: `key` que guarda la clave del nodo, `left` y `right` que los diccionarios de los hijos izquierdo y derecho.

Con respecto a las funciones, podemos comentar de su implementación que es casi todo directo a excepción de las siguientes funciones:

- `crearNodoA`: Crea un diccionario con la clave `key` a `None` y las claves `left` y `right` como diccionarios vacíos. Debido a lo comentado arriba, esta función no se puede hacer pública.
- `arbolVacio`: Devuelve un diccionario vacío.
- `insertar_r`
- `insertar_i`
- `buscar_r`
- `buscar_i`
- `sup2`
- `eliminar_r`
- `eliminar_i`

Como conclusión, esta implementación fue la más sencilla en cuanto a que la traducción fue directa.

5.2.3 Possible Enhancements