Design of Programming Languages

Iago Otero Coto - iago.oteroc
iago.oteroc@udc.es
Jorge Viteri Letamendía - j.viteri.letamendia
j.viteri.letamendia@udc.es

Python
OCaml
C
C++
Prolog

Contents

1 Python

1.1 Information

Compiler used: Python 3.5.2
O.S.: Linux Bash Shell in Windows 10 with Ubuntu 16.04.2 LTS (Xenial Xerus)
To execute: python3 abb_program.py

1.2 Introduction

Python is a high-level programming language for general-purpose programming. It is an interpreted language and belongs to multiple paradigms like imperative and object-oriented. It is also considered to be a scripting language and it is known for emphasizing readability.

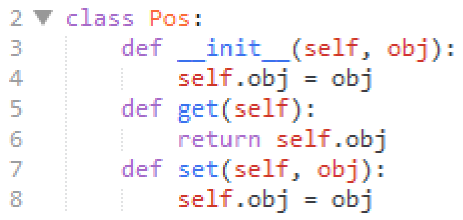
Python features a dynamic type system with an automatic memory-management. This is done by a cycle-detecting garbage collector.
Another important feature of this programming language is its late binding, which allows the list of methods of an object to be altered at runtime.

1.3 Analysis

The scripting and readability characteristics of Python, along with the fast edit-test-debug cycle and the absence of the compilation step, allow us to rapidly implement a solution to our problem.

However, the absence of pointers in Python will cause certain problems that we will solve by using object-oriented functionalities. More precisely, in the data structure definition, we will have to simulate pointers with the Pos class shown in Figure 1.

Apart from the one explained above, Python will not cause any other problem in the design or implementation of our solution.



```python
class Pos:
    def __init__(self, obj):
        self.obj = obj
    def get(self):
        return self.obj
    def set(self, obj):
        self.obj = obj
```

Figure 1: Pointer simulation with Pos class.

1.4 Design and implementation

As stated in the previous paragraphs, we will create a Pos class to simulate pointers. The two other classes that we will use are the Node and BinarySearchTree classes. We decided to model them in classes because it is quite straightforward and this way we will not need any workarounds since Python does not have records. Besides, BinarySearchTree is not exactly the same as tABB because instead of being the same as tPos, we have defined it as a different class with the atribute "root", which is the equivalent of tPos. This is a little different from the original code, but it is the closest we could get.

1.5 Details and disagreements

The only feature of Python that we are not fully taking advantage of, is the fact that you can define class methods. We only use the "__init__" methods, and the "get" and "set" methods of the Pos class.

Giving Python nature, we should be defining most of the functions as class methods to make the most of its object-oriented side. However, as our goal is to respect the desing and implementation decisions, we are going to define them as independent functions.

Apart from that, we have been able to use most of Python beneficial features.

2 OCaml

2.1 Information

Compiler used: The OCaml toplevel, version 4.02.3
O.S.: Linux Bash Shell in Windows 10 with Ubuntu 16.04.2 LTS (Xenial Xerus)
To compile:

```
ocamlc -c -w -8 abb.mli abb.ml abb_program.ml
ocamlc -o abb abb.cmo abb_program.cmo
```

To execute: ./abb

2.2 Introduction

OCaml, the main implementation of Caml, is a high-level, general purpose programming language that unifies functional, imperative and object-oriented paradigms. Its compiler employs static program analysis, which maximizes the performance of the resulting code.

OCaml also features type inference, parametric polymorphism and automatic memory management.

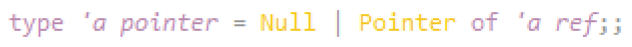
2.3 Analysis

Due to the fact that OCaml is a strongly typed functional language, it allows us to make concise and fast code. It also provides an interactive "read-eval-print" loop, which is convenient both for learning and for rapid testing.

Pointers do exist in OCaml. This can be a surprise for the "rookies" because, in most of the cases, pointers are used implicitly. Explicit pointers can be made by using one of the two built-in mutable data structures in OCaml: the ref. Ref is a primitive parameterized type which enables the use of imperative programming and, as explained before, it can be also used to define a general pointer type.

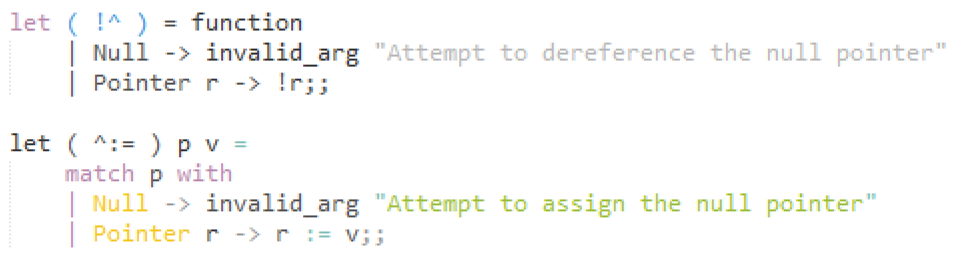
As per the general pointer definition, a pointer can be null or it can be also pointing to an assignable memory location, as shown in Figure 2.
In order to be able to read the designated value of the pointer and to write in its designated memory location, we will have to define a prefix operator (!^), and an infix operator (^:=) respectively. Figure 3 shows the definition of both operators.

Ultimately, we can define the allocation of a new pointer so that it will point to a given value as shown in Figure 4.



```
type 'a pointer = Null | Pointer of 'a ref;;
```

Figure 2: General pointer definition.



```
let ( !^ ) = function
    | Null -> invalid_arg "Attempt to dereference the null pointer"
    | Pointer r -> !r;;

let ( ^:= ) p v =
    match p with
    | Null -> invalid_arg "Attempt to assign the null pointer"
    | Pointer r -> r := v;;
```

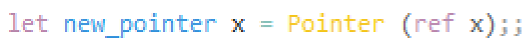
Figure 3: Dereferencing and pointer assignment definition.

2.4 Design and implementation

As explained before, we will be using generic pointers to simulate as much as possible the Pascal code. This will lead us to the following type definition shown in Figure 5. Due to the characteristics of OCaml, this is an excessively complicated way of implementation of a Binary Search Tree. Given that, in the ”Details and disagreements” section we will show a better way to do this that does not resemble the Pascal code that much. Apart from that, the rest of the code is quite self-explanatory and does not differ from the original.

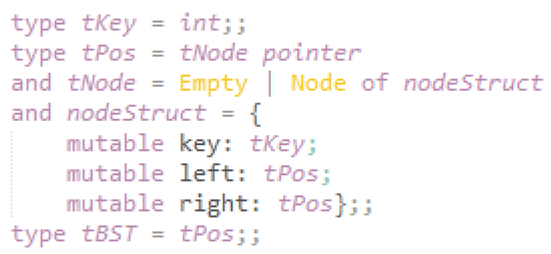
2.5 Details and disagreements

To make the best of OCaml programming language, we can define a more simplified definition of Binary Search Tree without pointers as shown in Figure 6. This allows us to implement a fast and easy way to insert keys in the tree (Figure 7). The issue with this implementation is that we cannot really translate the iterative version of insertKey and removeKey. We would have to return a different tree as we cannot change the one that we are receiving. Moreover, we would have to do some workarounds with Ref types even though we are not using it for the types definition. To sum up, it can be done, but it would move us away from both OCaml's beneficial features and the design and implementation of Pascal's code.



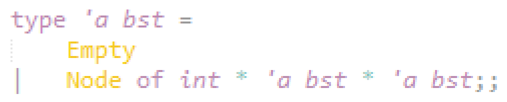
```
let new_pointer x = Pointer (ref x);;
```

Figure 4: Allocation of a new pointer definition.



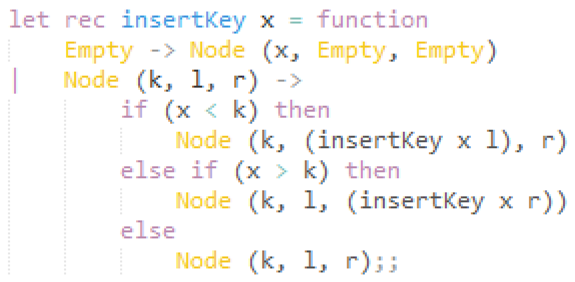
```
type tKey = int;;
type tPos = tNode pointer
and tNode = Empty | Node of nodeStruct
and nodeStruct = {
    mutable key: tKey;
    mutable left: tPos;
    mutable right: tPos};;
type tBST = tPos;;
```

Figure 5: Type definition.



```
type 'a bst =
    Empty
|   Node of int * 'a bst * 'a bst;;
```

Figure 6: Simple definition.



```
let rec insertKey x = function
    Empty -> Node (x, Empty, Empty)
|   Node (k, l, r) ->
        if (x < k) then
            Node (k, (insertKey x l), r)
        else if (x > k) then
            Node (k, l, (insertKey x r))
        else
            Node (k, l, r);;
```

Figure 7: Simple definition.

3 C

3.1 Information

Compiler used: gcc 5.4.0
O.S.: Ubuntu 16.04 LTS
To compile: gcc -o abb Abb_functions.h Abb_functions.c Abb_program.c
To execute: ./abb

3.2 Introduction

C is a programming language that belongs mainly to the imperative paradigm and it follows the Von Neumann architecture. Its type checking is static and its memory management includes three types of allocation:
Static allocation (which applies to global variables, file scope variables, and variables with the static qualifier), automatic memory allocation (which is used for non-static variables defined inside functions) and dynamic memory allocation. This last type of allocation means that the allocation and deallocation of "objects" is done manually by instructions included in a library (stdlib.h in this case).

3.3 Analysis

The greatest advantage in this case is that C follows the imperative paradigm like Pascal. This means that translating the code is not very complicated as it only requires some simple changes in the syntax and type definition.

Another advantage of C is the use of pointers, which make the implementation of the tree very straightforward. On the other hand, this can lead to problems with memory leaking if we are not careful enough.

3.4 Design and implementation

Only some changes were made to translate the code from Pascal to C due to the reasons explained before, so there are not any design decisions to explain.

3.5 Details and disagreements

In this case, once again, due to the similarities between Pascal and C, we did not encounter any situation in which we were not taking advantage of the beneficial features of C.

4 C++

4.1 Information

Compiler used: g++ 5.4.0
O.S.: Ubuntu 16.04 LTS
To compile: g++ -o abb Node.h Abb_functions.h Abb_functions.cpp Abb_program.cpp
To execute: ./abb

4.2 Introduction

C++ is a generic purpose language that follows a multi-paradigm (Object Oriented, Imperative and Functional) and, like C, its type checking is static and has the same types of memory management.

4.3 Analysis

Due to the fact that C++ is an extended version of C, it has, in general terms, the same advantages and disadvantages but including object-oriented programming.

4.4 Design and implementation

As a consequence of the previous point, in order to differentiate these two languages, we decided to exploit some of the object-oriented benefits. We defined the node type as a class, even though this pushes us away from the original implementation. In addition, there are a few changes we had to do concerning pointers but nothing too significant.

4.5 Details and disagreements

Although we used some of the object-oriented features, we did not take advantage of the benefits of defining class methods. This was done to maximize the similarities with the original code.

5 Prolog

5.1 Information

Compiler used: SWI-Prolog version 7.2.3 for amd64
O.S.: Linux Bash Shell in Windows 10 with Ubuntu 16.04.2 LTS (Xenial Xerus)
To execute: swipl -f abb.pl < abb_program.txt

5.2 Introduction

Prolog is a general purpose programming language and belongs to the logic paradigm. As a declarative language, its logic is expressed in terms of relations, and it is represented as facts and rules. In terms of memory management, its garbage collector will be invoked automatically when necessary, but it can also be called manually.

5.3 Analysis

One of the main features of Prolog is that it is not a typed language. This means that it does not do any type checking by itself and you can use any term as a parameter for a given predicate. Consequently, the prototyping is really fast and the code is very compact.

On the other hand, it is very easy to make a mistake due to lack of prevision, as the compiler will not help you to find it.

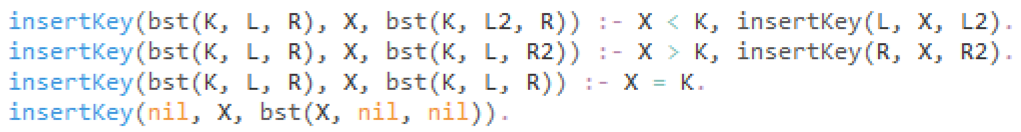
5.4 Design and implementation

Our Prolog code will not resemble the Pascal code too much since the paradigms are completely different. Nonetheless, if we abstract the design from the implementation, we can get closer. For example, we can translate the recursive version of insertKey into four rules preserving the same idea (Figure 8). However, the iterative methods are so far away from Prolog's nature that we will not be able to translate them any way.

Regarding the representation of the Binary Search Tree type, we can not explicitly define types in Prolog (at least not without external libraries), so we just used the term bst(K, L, R) as the parameters of the functions to represent a tree whose root is K and whose left and right children are L and R.

It is also worth mentioning that we do not modify the "input tree". Instead of this, we add an extra parameter to the function which will hold the desired response. This way, the Prolog engine will attempt to find a resolution to the query and fix a value in that said parameter to satisfy the query.

As a final note, given that Prolog works with queries, we have built a big query that tests the same things as the provided code. This works fine but, at the end, it prints the values of the terms that we used in the query. This is something we actually do not need to know.



```
insertKey(bst(K, L, R), X, bst(K, L2, R)) :- X < K, insertKey(L, X, L2).
insertKey(bst(K, L, R), X, bst(K, L, R2)) :- X > K, insertKey(R, X, R2).
insertKey(bst(K, L, R), X, bst(K, L, R)) :- X = K.
insertKey(nil, X, bst(X, nil, nil)).
```

Figure 8: insertKey definition.

5.5 Details and disagreements

There is not too much to say about the unused beneficial features of Prolog. It kind of forces you to use them, so we are not missing a lot.