TP 1 : Classes and Objects

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Exercise I : Visibility (LISP)

Q1 – Create a class `player`

Q2 – Add slots to your class matching the name and the age of a player ; provide getters for these slots.

Q3 – Define an instantiation function `make-player` with named parameters for class `player`.

Q4 – Add an integer field `count` that counts the number instances created. You will need to change your function instantiation so that it increments the value of *count* to each call.

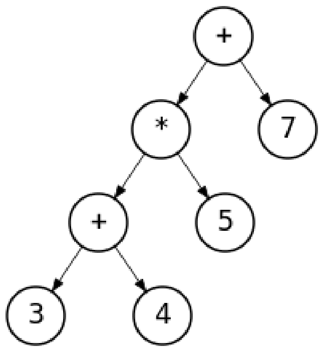
Q5 – Add to the class a slot `team` of type `boolean`. We want to guarantee parity in such a way that that the number of instances having a `team` field at `true` is the same (within 1) as the number of instances having the `team` field set to `false`. How to do ?

Q6 – Define a function `print-player` that displays on the standard output the information relating to an instance of the `player` class.

Q7 – Create some instances of the `player` class and display their information.

Exercise II : Design Pattern *Composite*

This design pattern (*design pattern*) allows to design a recursive tree structure, allowing to implement with the same interface for leaves and nodes so that they are handled in the same way. This type of representation is called tree an abstract syntax tree (AST) where Each node of the tree designates a construction appearing in the text. By example the expression $(3 + 4) * 5 + 7$ can be represented by the AST following :



In this exercise, we will use the design pattern *composite* to implement an expression evaluator arithmetic. We will first assume that our expressions arithmetic are either constants or binary operations among $\{+, -, *, /\}$.

Q1 – Create a class *Constant* to model constants integers with a *value* slot.

Q2 – Create a class *Binop* to model operations binaries. A binary operation has two operands (which are themselves arithmetic expressions) and an operator represented by a character string.

Q3 – Define functions *plus*, *minus*, *mul* and *div* which take two expressions as parameters and return an instance of the *Binop* class. Define a function *cst* which takes an integer as a parameter and builds an instance of *Constant*.

Q4 – Create two *evaluate* functions, one that takes a instance of the *Constant* class and the other an instance of the class *Binop* as a parameter. They should return the value of the corresponding arithmetic expression. For binary operations you can use the following auxiliary function to choose the right operation to apply :

```
(defun op-to-lambda (op)
  (if (string= op "+") '+
    (if (string= op "*") '*
      (if (string= op "-") '-
        (if (string= op "/") '/
          (error "unrecognized operator")))))
```

Q5 – Verify that `(evaluate (plus (cst 2) (mul (cst 4) (cst 10))))` gives 42.

Exercise III : Functions as Objects

We are now going to add functions to our language. For that we want to be able to represent three new types of expression : functions, function calls and variables. Modeling possible is to refer to the arguments passed to a function by their position (and not by their name), as in bash, by example :

```
identity() {
  echo "$1"; #print first argument
}
```

Q1 – For this add three classes :

— a class *fun* with a slot *body* which will be a expression,

— a *argument* class with an entire *index* slot,

— a class *call*, for function calls, with a slot *fun* which will be an instance of *fun* and a slot *args* which will be a list of expressions.

Q2 – Create constructors *make-fun*, *make-arg* and *make-call* for these classes. For example, the square function can be declared as follows :

```
    (defvar square (make-fun :body (mul (make-arg 0) (make-arg 0))))
```

Q3 – To be able to evaluate function calls, we need to provide a data structure that will store the arguments value while preserving the order in which they are passed. We will therefore modify the *evaluate* methods of constants and binary operations to add a *context* argument that will play this role. It is not no need to modify the body of these methods.

Q4 – Add two functions *evaluate* : one that takes a instance of the class *argument* and the other an instance of the class *call* as a parameter, and which return the value of the expression according to the following rules :

— the evaluation of the n-th argument is to retrieve the n-th value of *context*.

— evaluation of a function call consists of evaluating the arguments, populate the evaluation context with the list of results and evaluate the body of the function being called. Forthe evaluation of the arguments, you can use the function *mapcar* which takes a lambda f and a list of values '$(a_1 \ldots a_n)$ and builds the list '$((f\ a_1) \ldots (f\ a_n))$ with the results returned by f, as in the example below.

Example | *(mapcar (lambda(x) (+ x 1)) '(1 2 3)) = '(2 3 4)*

Q5 – Verify that

— `(evaluate (plus (make-arg 0) (make-arg 1)) (list 1 2))` gives 3

— `(evaluate (make-call square (list (cst 7))) ())` gives 49.