A Class System for Common Lisp

António Menezes Leitão

April 2017

1 Introduction

In this project, we want to challenge you to implement a Class System for Common Lisp, but without using using any predefined CLOS features or the `defstruct` facility.

The Class System automatically generates *getters* from a class definition but it does not need to generate *setters*. This means that class instances will be *functional*, that is, after created, the instance cannot be modified.

In the next subsections we will explain the capabilities of the Class System.

1.1 Class Definitions

The first operation to consider is the definition of a class. Here is an example:

```
(def-class person
  name
  age)
```

And here is an example of its use:

```
> (let ((p (make-person :name "Paulo" :age 33)))
    (person-age p))
33
```

Given that we must use a restricted subset of Common Lisp, we will need to translate (some people would say "expand") the previous class definition into something that could look like the following:

```
(progn
  (defun make-person (&key name age)
    (vector name age))
  (defun person-name (person)
    (aref person 0))
  (defun person-age (person)
    (aref person 1)))
```

Note the presence of the *constructor* `make-person` and *getters* `person-name` and `person-age`.

However, the previous *expansion* is not sufficient, because we also want to support the automatic generation of a *recognizer* that allows us to test the instance:

```
> (let ((a (make-person :name "Paulo" :age 33))
        (b "I am not a person"))
    (list (person? a) (person? b)))
(T NIL)
```

Moreover, as the following sections will show, our Class System has other sophisticated capabilities that require a more complex implementation.

1.2 Inheritance

Inheritance allows us to define hierarchies such as:

```
(def-class (student person)
  course)
```

When a class inherits from another class, its definition must include the slots of the inherited class.

It is now possible to do things like the following:

```
> (let ((s (make-student :name "Paul" :age 21 :course "Informatics")))
    (list (student-name s) (student-course s)))
("Paul" "Informatics")
```

1.3 Subtyping

One important property of inheritance is that it makes the *getters* of the superclass applicable to the instances of subclasses. As an example, consider:

```
> (let ((s (make-student :name "Paul" :age 21 :course "Informatics")))
    (list (person-name s) (student-course s)))
("Paul" "Informatics")
```

Moreover, subtyping allows for the *recognizer* of a class to be applicable to instances of its subclasses. For example:

```
> (let ((p (make-person :name "John" :age 34))
        (s (make-student :name "Paul" :age 21 :course "Informatics")))
    (list (person? p) (student? p) (person? s) (student? s)))
(T NIL T T)
```

1.4 Multiple Inheritance

The Class System also supports multiple inheritance. Here is one example:

```
(def-class sportsman
  activity
  schedule)

(def-class (ist-student student sportsman))

(let ((m (make-ist-student :name "Maria" :course "IA" :activity "Tennis")))
           (list (ist-student? m)
                 (student? m)
                 (sportsman? m)
                 (ist-student-name m)
                 (person-name m)
                 (sportsman-activity m)
                 (ist-student-activity m)))
(T T T "Maria" "Maria" "Tennis" "Tennis")
```

2 Goals

The main goal of this project is the implementation of the Class System, using the syntax and semantics previously described.

More specifically, you need to implement the macro `def-class` and auxiliary operations that, from a class definition, generates the functions that implement the *constructor*, *recognizer*, and *getters* of the class.

2.1 Extensions

You can extend your project to further increase your grade above 20. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections.

Some interesting extensions are:

- Automatic generation of *setters*.

- Default initialization values for slots.
- Single-dispatch methods.
- Generic functions and multiple-dispatch methods.
- Meta-classes.
- Meta-object protocols.

3 Code

Your implementation must work in any Common Lisp implementation, for example, SBCL.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

We expect all the required functionality to be available in the `COMMON-LISP-USER` package. If you don't define new packages, then you don't need to do anything to fullfil this requirement.

4 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 15-minute slot, should be centered in the architectural decisions taken and may include all the details that you consider relevant. You should be able to "sell" your solution to your colleagues and teachers.

5 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named as `project.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing:

- the source code, within subdirectory `/src`
- the slides of the presentation, in a file named `p2.pdf`
- a `.lisp` file `load.lisp` file that, when loaded, compiles and loads the code into the Common Lisp implementation.

The only accepted format for the presentation slides is PDF. This PDF file must be submitted using the Fénix Portal separately from the ZIP file and should be named `p2.pdf`.

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner workings of the developed project, including demonstrations.

The public presentation of the project is a compulsory evaluation moment. Absent students during project presentation will be graded zero in the entire project.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code and the slides must be submitted via Fénix, no later than 23:59 of **May**, **19**.

The presentations will be done during the classes after the deadline. Only one element of the group will present the work and the presentation must not exceed 15 minutes. The element will be chosen by the teacher just before the presentation. Note that the grade assigned to the presentation affects the entire group and not only the person that will be presenting. Note also that content is more important than form. Finally, note that the teacher may question any member of the group before, during, and after the presentation.