(LINFO2335 — Assignment Module 2)
Functional Programming Mission:
Simulating an object-oriented language in Scheme

K. Mens, J. Liénard, O. Goletti

March 2024

Introduction

Objectives

During the second module of this course, you have been introduced to the LISP language dialect Scheme and the functional programming paradigm. Since this is a course about programming paradigms, in this second project we ask you to simulate the object-oriented programming paradigm through functional programming in Scheme.

You will do this in a stepwise fashion, going from a simple solution that allows you to emulate objects in terms of function closures carrying their own state, to more elaborate solutions that support class definition, object creation, simple inheritance, super calls to parent class, references to self with late binding, etc.

We hope that by carrying out this project, you will gain a better feeling of the functional programming style and obtain deeper insights in the concepts underlying object-oriented programming as well.

Organisation

Since we advocate *learning by doing*, this project will need to be carried out by pairs of students. This will allow you to confront your ideas and learn from each other. We do expect each student to contribute equally to the project and if needed may verify this through oral interviews at the end of this project.

You have to hand in your final solution at the end of week 10, on Friday 26 April before midnight. Your submission shall contain properly documented files, containing your implementation of each of the steps, as well as sample code illustrating the execution of these steps. For each step, we thus expect a separate file with your solution and some exemplifying code. We should be able to easily understand and run each single file independently and receive a nice output illustrating the implemented functionality. To avoid incompatibility between Scheme dialects you shall respect the R5RS language standard strictly.

We will download, read and try out your code in the weeks that follow and optionally invite you for an oral interview if we need additional information (but your code should be self-contained so that we could evaluate it even without such an interview).

Some time will already be devoted to this project during the two last lab sessions of the Scheme module of the course. However, we expect of you to be proactive and show us what you can do, going further than the present guide if you can. This guide thus serves only as a set of minimal requirements to get a passing grade.

Grading

The grading for this project will be based upon our comprehension and the executability of your submitted code, as well as on your capacity to explain, elaborate and reason about your code, based on an interview with the course assistant and/or professor. During this interview, you will have to show us that you understand the different language concepts seen throughout the course. Both students should be able to understand, explain and answer questions related to their code (or slight variations of it) and the language concepts used in it.

Implementing the first four steps of this guide (and being able to defend your choices during the interview) will be sufficient to get a passing grade. We expect you to complete the fifth step as well, which is slighlty more challenging but will allow you to get an even higher score. Going beyond what is explained in this guide will allow you to reach a top score.

Step 0: objects as dispatchers returning values stored in their closure

In the slides of the theory course, we saw an interesting way to define your own *pair* (i.e., cons-cells) representation in terms of a `dispatch` procedure. The mechanism uses a higher-order function `my-cons` that returns a `dispatch` function that will match specific messages to access either the `my-car` or the `my-cdr` of the pair.

```
(define (my-cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (display "Argument not 0 or 1"))))
  dispatch)
(define (my-car z)
  (z 0))
(define (my-cdr z)
  (z 1))
(my-car (my-cons 1 2)) ; returns 1
(my-cdr (my-cons 1 2)) ; returns 2
```

In this first step of the Scheme assignment, we want you to use a similar mechanism to emulate a `point` class as a constructor function that creates an object which is essentially a dispatch function that knows how to respond to messages. The state is passed with the constructor and stored in the closure of the dispatcher so that every object created like this carries its own state around in the closure of its dispatcher.

Your `point` class, once implemented, should allow you to run code similar to this:

```
(define p (point 1 2))
(p 'getx) ; 1
(p 'gety) ; 2
(p 'type) ; point
(p 'info) ; (point 1 2)
```

The structure of your code will be very similar to how `my-cons` was implemented, albeit that the dispatcher of your `point` constructor will understand more messages (namely `'getx`, `'gety`, `'type` and `'info`).
Additional question: are you able to implement the `'info` message in terms of a call to the other messages `'getx`, `'gety` and `'type`?

Step 1: objects as dispatchers returning functions

Your solution of the previous step probably falls short in being able to handle messages that take additional parameters, such as an `'add` (to pairwise add the coordinates of two points) or a `'setx!` message (to set the x-coordinate to new value).

Adapt the solution by making the dispatcher function return, for each possible message, a function that knows how to treat that message. Your adapted `point` class constructor, once implemented, should allow you to run code similar to this (and more):

```
(define p1 (point 1 2))
(define p2 (point 3 4))
((p1 'getx)) ; returns 1
((p1 'gety)) ; returns 2
((p2 'getx)) ; returns 3
((p2 'gety)) ; returns 4
(define p ((p1 'add) p2)) ; returns a new point p
((p 'info)) ; returns (point 4 6)
((p 'foo))  ; should display "Message not understood" error
((p1 'setx!) 5)
((p1 'getx)) ; returns 5
```

Note the extra pair of parentheses around each message call because the dispatcher messages don't immediately return values, but rather functions that can evaluate those values. On the positive side, this extra indirection does allow us to simulate messages that take parameters now. For example, `(p1 'setx!)` would return an internal function that knows how to assign a new value to the x-coordinate, and `((p1 'setx!) 5)` would then call that function to set the x-coordinate to 5.

Remark: In order to make your code feel more object-oriented, you can name your dispatch procedure `self` instead of `dispatch`.

Step 2: defining a message sending operator

Now that we have simulated classes and objects through dispatcher functions carrying their own lexical scope, let us improve the syntax a bit and create a `send` operator, so that we can write things like `(send p 'getx)` instead of this ugly syntax `((p 'getx))` with two pairs of parentheses, and `(send p1 'add p2)` instead of `((p1 'add) p2)`.

This `send` operator should check first if the receiver object is indeed an appropriate receiver object (i.e., a dispatch procedure) and look up if the message sent corresponds indeed to a message that can be understood by this receiver object.

After this second step, you should be able to execute the following code:

```
(define p1 (point 1 2))
(define p2 (point 3 4))
(send p1 'getx) ; 1
(send p1 'gety) ; 2
(send p2 'getx) ; 3
(send p2 'gety) ; 4
(define p (send p1 'add p2))
(send p 'info) ; (point 4 6)
(send 'not-a-point 'info) ; should display "Inappropriate receiver object"
(send p 'foo) ; should display "Message not understood"
(send p 'bar 2) ; should display "Message not understood"
(send p1 'setx! 5)
(send p1 'getx) ; returns 5
```

Remark: Making sure that informative messages are displayed when calling `send` with inappropriate objects, messages or numbers of arguments will make it easier to understand your errors and debug your code.

Step 3a: cleanup

This intermediate cleanup step might not be necessary if your implementation of the previous step already has the expected structure (see below).

Method definitions It would be nice if, just like in real object-oriented programs, you would define methods explicitly in the class as separate named functions. Even though the dispatch mechanism would still be present in your implementation of classes, it should be separated and abstracted from the actual implementation of the functions called by that dispatcher. The only thing the dispatcher would do is to lookup messages and return the functions corresponding to those messages and defined elsewhere in the class. This will lead to a much more object-oriented look-and-feel for class writing.

For example, a `bank-account` class could look like this:

```
(define (bank-account balance)
  (define (get-balance) balance)
  (define (withdraw n) (set! balance (- balance n)))
  (define (type) 'bank-account)
  (define (info)
```

```
    ; ...
   )
   ; dispatch code
   ; ...
   ; return dispatch procedure
)
(define ac (bank-account 100))
(send ac 'withdraw 50)
(send ac 'info)
```

send and method-lookup procedures The `send` procedure with signature `(send receiver message . args)` and its error management should be abstracted outside of the class definitions at this point. Your `send` procedure should first lookup in the receiver object the actual procedure corresponding to the message sent (let us call this the method).

We also suggest that you create a separate method-lookup procedure with signature `(method-lookup receiver message)` to perform this lookup. (Separating this lookup mechanism returning the method in a separate produre will become useful in the next step when we will need to delegate the lookup to super classes.) Once this method has been found, your `send` procedure should **`apply`** that method to the argument list passed along to the `send` procedure.

Step 3b: introducing inheritance

In this third step we will add the notion of class inheritance. This means that each class will now carry in its state a reference `super` pointing to its (unique) parent class. The dispatch or method lookup mechanism will need to be adapted so that when a message is called on a receiver object and the class of that object does not define that method, the message lookup will look in the superclass instead (and so on).

For this to work, we will also need to introduce a new class `object` which will serve as the root of the inheritance hierarchy. (Just like in Java there is a class Object that serves as direct or indirect superclass for all classes.) This root class is where the message lookup will eventually stop. This class will only implement the `'type` method. In the `object` class, `super` will be bound to `'nil`, to indicate that it has no further super class.

When defining a new class, like `point`, you should indicate from what parent class it directly inherits. If it doesn't inherit from any other class that you defined, you should make it it inherit from `object`. (This is the case for the class `point`.)

Once you have defined the `point` class you can use your new inheritance mechanism to implement a `color-point` class which inherits from `point`. In addition to the point's coordinates it will keep a `color` attribute in its state. Redefine the appropriate methods of class `color-point` where needed, and delegate to the super class what you can. Adding up two `color-point` objects should yield a new `color-point` with coordinates being summed and with as color the color of the receiver.

You should now have `color-point` class which inherits from `point` which inherits from `object`.

You should now be able to execute the following code:

```
(define o (object))
(send o 'type) ; object
(send o 'foo)  ; should display "Message not understood"

(define p1 (point 1 2))
(define p2 (point 3 4))
(send p1 'getx) ; 1
(send p1 'gety) ; 2
(send p2 'getx) ; 3
(send p2 'gety) ; 4
(define p (send p1 'add p2))
(send p 'info) ; (point 4 6)

(define cp (color-point 5 6 'red))
(send cp 'type)         ; color-point
(send cp 'getx)         ; 5
(send cp 'gety)         ; 6
(send cp 'get-color)    ; red
(send cp 'info)
; depending on your implementation this could result in
; (point 5 6 red) or (color-point 5 6 red)
; both of these are OK at this step, try to understand why
; more about this in step 4
(define cp-1 (send cp 'add (color-point 1 2 'green)))
(send cp-1 'type)       ; color-point
(send cp-1 'getx)       ; 6
(send cp-1 'gety)       ; 8
(send cp-1 'get-color)  ; red
```

Step 4: dynamic binding of `self`

When delegating a call to the super class, your current implementation may not always handle the method lookup of self-sends correctly yet. Why is that? Because calls delegated to a method defined in the super class are not executed in the context of the child class.

Ideally, we want to achieve something like this:

```
(define (parent-class)
    ; inherits from object
    ; ...
    (define (method1) (send self 'method2))
    (define (method2) 'parent)
    ; ...
   )
   ; ...
)
(define (child-class)
    ; inherits from parent-class
```

```
    ; ...
    (define (method2) 'child)
    ; ...
   )
   ; ...
)
(define d (new child-class))
(send d 'method2) ; child (this is obvious)
(send d 'method1) ; child (dynamic binding of self)
```

This desired behaviour (which we will ask you to implement in this step) is the principle of *dynamic or late binding of self* in object-oriented programming. When sending a message `'method1` to some receiver object `d` which gets looked up in a super class, and that super method `method1` calls another message `'method2` on `self`, then that `method2` should be looked up starting from the class of the *receiver* object `d` again, rather than remaining in the scope of the super class.

Until now, the dispatch procedure has no knowledge of what class `self` is bound to. We therefore need to implement a mechanism so that `self` gets dynamically bound to the class of the original receiver of a message send. And whenever method lookup delegates to a method defined in a super class, the `self` reference of that superclass should also be set to the subclass, so that the lookup starts again from the subclass in case of self-sends.

`new` constructor procedure In order to achieve this dynamic binding of self, we will first create a `new` procedure that will handle the creation of any class instance. At this point, you should be able to write this procedure, it is just an indirection like the `send` and `method-lookup` procedures from last step.

Binding `self` to the receiver The next step is to change `new` so that it explicitly binds `self` to the instance being created. But it should do so recursively for all of its super classes as well, so that these know what self to call back in case of a self-send.

One way of doing so is to add a special method named `'set-self!` to each class, which binds `self` to a given instance and then recursively calls the same `'set-self!` method on the superclass to bind its `self` to that same instance. This entire process is initiated by the `new` procedure.

For now, it is okay if you duplicate the code of this `'set-self!` method in each class. Once you get the code working in this way, we leave it for you to think about how this duplicated code could be avoided.

After this step, you should be able to execute this code:

```
(define cp (new color-point 5 6 'red))
(send cp 'type)          ; color-point
(send cp 'getx)          ; 5
(send cp 'gety)          ; 6
(send cp 'get-color)     ; red
(send cp 'info)          ; (color-point 5 6 red)
```

This time, make sure that the call to `(send cp 'info)` makes a super call to the unmodified `'info` method in `point` (like proper inheritance).

The magic should happen when the `'type` implementation used is the one of `color−point`. This time, it should print `(color−point 5 6 red)`.

Step 5: generating dispatchers with hygienic macros

During the course, we have briefly mentioned how to make hygienic macros in scheme using `define−syntax`. For now, for each class, you probably have written the dispatchers by hand. In object-oriented programming languages, you don't have to write your dispatchers, the compiler will do it automatically according to the declared method.

In this final step, we will create a macro called `define−class` which should define a new class and generate the dispatcher without having to list explicitly all the method names in the dispatcher.

Your macro should bind the name of your methods to their lambda-expression in a list using **`quasiquote`** and **`unquote`**. Then, you can create a dispatcher function that will check if the message that it receives matches any of the names of your methods using `assq`. Here is an example of what your binding list could be and how `assq` should work:

```
(define dispatcherlist `(
  (plus . ,(lambda (x y) (+ x y)))
  (minus . ,(lambda (x y) (− x y)))
  ))
(assq 'plus dispatcherlist) ;(plus . #<procedure:...>)
(cdr (assq 'plus dispatcherlist)) ;#<procedure:...>

(let ((m (cdr (assq 'plus dispatcherlist))))
  (display (m 1 2))) ; will display 3
```

In the example above, the ` before the parentheses is a **`quasiquote`** and the **`,`** before the `methodbody` is the **`unquote`** as explained here. In essence, quasiquote is similar to quote but allows to still evaluate subexpressions inside the expression you are quoting, by unquoting them with **`,`** as for example in:

```
(define a 1)
(define b 2)
`(a b ,(+ a b) ,(− a b) (* a b))
; returns (a b 3 −1 (* a b))
```

This mechanism of quasiquote and unquoting is often used when using macros to construct symbolic expressions that need to be *partially* evaluated.

Now, your class declaration should be close to something like this:

```
(define−class (bank−account balance)
  ; init self and super
  ; ...
    (define (get−balance) balance)
    (define (withdraw n) (set! balance (− balance n)))
    (define (type) 'bank−account)
    (define (info) ...)
    ; ...
  ; no need to define a dispatcher function manually,
```

```
; this should be generated by your define-class macro
)
```

With such class declarations, you are close to how you would write things in a real object-oriented programming language. The Scheme macro `define-class` that you need to write will generate the boilerplate code of your dispatcher function at compilation time. This boilerplate code will create a list of bindings of methods to their lambdas and use `assq` to get the lambda-function corresponding to the message received by the dispatcher.

Step 6: generating even more boilerplate code

Note that, in this fifth step, we only ask you to generate the dispatcher code with a macro. But you may have noticed that other parts of your class definitions are the same for all your classes, such as the initialisation of `self` and `super`. These could be added to the boilerplate code generated by your macro as well. We leave this as an additional bonus step 6 for you to complete if you are aiming for that top score.

Submission

Here are some final instructions regarding the submission of this assignment:

- You don't need to write a separate report; submitting your code clearly commented is sufficient (see below).
- We expect 6 (max. 7) files containing the Scheme code corresponding to the different steps of your assignment (step 0, step 1, step 2+3a combined, step 3b, step 4, step 5 and optionally a final step 6 with bonus extensions).
- Each file should clearly mention the name of both group members that worked on this file.
- Each file should start with a detailed comment section that explains how you implemented this step, what were the key design decisions underlying your solution and so on.
- Each file should end with sample code illustrating the execution of this step by providing a nice output illustrating the implemented functionality. (Include in comments the output we should expect when running your code.)
- Each file should be executable as a Scheme file in the latest version of DrRacket using the R5RS language setting.
- Your Scheme code should be properly documented: good names of functions and variables, good formatting and indentation, and comments explaining the code where needed, so that the source code is self-contained and understandable without a separate report.
- Your final code should be submitted on Moodle on Friday 26 April before midnight. We will download it on Saturday 27 April to read and evaluate it.