Tutorial Problem Set 5

CS 152 – Abstractions and Paradigms in Programming

The purpose of this assignment is to familiarize you with the object-oriented programming style.

1. Let us design a processor, which will be called cs152-2018. Here is a sample program in the assembly language of cs152-2018. The comments will tell you of what the instructions of the machine do. The program is represented as a list of instructions.

```
(define sample-prog (list
    (list 'load 'r2   15)   ;Assign to register r2 the number 15
    (list 'load 'r1    5)   ;Assign to register r1 the number 5
    (list 'add  'r1  'r2)   ;Add r1 and r2, the result goes to r1
    (list 'load 'r3    2)   ;Assign to register r3 the number 2
    (list 'incr 'r3)        ;Increment contents of register r3 by 1
    (list 'mul  'r3  'r2)   ;Multiply r3 and r2, the result goes to r3
    (list 'add  'r1  'r3)   ;Add r1 and r3, the result goes to r1
    (list 'store  1  'r1))) ;Store r1 in memory location 1
```

   You could find the effect of running the program by:

```
(send cs152-2018 execute sample-prog)
```

   The output shows the registers and the memory after execution of each instruction. Note carefully how they are represented.

   To continue the story, you create cs152-2018 from a class called processor%. As explained in the class processor% is the abstraction of all processors that we want to model. Such processors:

   (a) Have a load and a store instruction with a fixed meanings.

   (b) Have other instructions whose names and functionalities can vary from processor to processor. We have assumed that such instructions necessarily operate from registers.

   (c) Have memory whose size can vary from processor to processor.

   (d) Have a bank of registers whose names and number can vary from processor to processor.

   (e) Have the capability of executing the instructions of a program, given the interpretations of the non-load and non-store instructions.

   Assume that you have defined processor%. Then here is a skeleton of how cs152-2018 is created using processor%. You have to fill in the missing parts:

```
(define cs152-2018
  (new processor%
       [mem-size 32]
       [reg-names (list `r1 `r2 `r3 `r4)]
       [inst-set (list  (list 'add  (lambda (val1 val2)
                                      (+ val1 val2)))
                        (list 'mul  ...)
                        (list 'incr ...))]))
```

We shall now design a class for the register-bank and another for memory. Since they are similar, we shall inherit them from a class called `storable%`. Here is `storable%`.

```
(define storable%
  (class object%
    (super-new)
    (define/public (read) (error "Should be overridden"))
    (define/public (write) (error "Should be overridden"))
    (define/public (print) (error "Should be overridden"))))
```

This class forces any class that inherits from it to define the methods `read`, `write` and `print`. Now define the classes `register-bank%` and `memory%` by inheriting from `storable%`. [1].

Now all that remains is to define the class `processor%`. Do it by filling the following template:

```
(define processor%
  (class object%
    (super-new)

    ; Do the initializations here

    (define/public (execute prog)
      (if (null? prog)   "Done"
             ...
             ...
            (execute (cdr prog))))))
```

When you run a program using `cs152-2018`, or for that matter any other processor instantiated from the class `processor%`, the memory contents and the values of the register should be displayed after each step in the format shown.

2. In this question you have to design a banking system consisting of four classes: An `account` class, a `joint-account` class, a `credit-card` class and a `timer` class. Their behaviour is illustrated by the following commentary.

[1] Yes I know that none of the instruction require to read from memory, but think of the processor `cs152-2019` that is going to come out around the same time next year. It is possible that it may have such an instruction

```
(define this-timer (new timer%))
```

There is only one `timer%` object called `this-timer`. The purpose of the timer is to trigger certain activities at designated points of time called *ticks*. For instance, on a `process-accounts-tick`. interests are paid to all accounts. Similarly, uncleared dues on a credit card are penalized on a `process-credit-card-tick`.

```
(define my-account (new account% (passwd "amitabha")
                                 (balance 1000)
                                 (interest-rate 0.05)))

(define my-card (new credit-card% (acct my-account)
                                  (passwd "amitabha")))

(define another-account (new account% (passwd "abcd")
                                      (balance 5000)
                                      (interest-rate 0.1)))

(define another-card (new credit-card% (acct another-account)
(passwd "abcd")))

(define jnt-account (new joint-account% (account my-account)
(passwd "amitabha") (new-passwd "xyz")))
```

The meanings of these are obvious. Two accounts and two credit cards are created. Apart from `withdraw`, an account should also have a method called `deposit` and other methods appearing in the commentary. Each credit card is tied to an account and is created using the password of the account. A joint account is also created.

```
(send my-account withdraw 200 "amitabha")
(send my-account show "amitabha")
> 800


(send jnt-account withdraw 200 "amitabha")
>"Incorrect Passwd"
```

The joint-account cannot be operated using the original password.

```
(send jnt-account show "xyz")
>800

(send my-account pay-interest)
send: no such method: pay-interest for class: account%
```

There is a method `pay-interest` in the `account%` class. But it cannot be accessed from outside the class.

```
(send this-timer process-accounts-tick)
(send my-account show "amitabha")
>840.0
```

The time has come to pay interest of 5% on outstanding balance. Ideally the timer should generate the tick. But to keep things simple, we generate the tick from outside and inform the timer.

```
(send my-card make-purchase 100)
```

I make a purchase of 100 rupees.

```
(send this-timer process-credit-card-tick)
```

Oops. I have not cleared my credit card dues. And the `process-credit-card` tick has arrived. So I have to pay a penalty. Let me try to clear my dues quickly and see whether this works.

```
(send my-card clear-outstanding-amount)
(send my-account show "amitabha")
>720.0
```

No luck. I had to pay the penalty of 20% on my credit-card dues of 100 rupees.

```
(send my-card make-purchase 50)
(send another-card make-purchase 1000)
(send another-card clear-outstanding-amount)
(send this-timer process-credit-card-tick)
(send my-account show "amitabha")
>720.0
```

Once again I have made a purchase and did not clear my dues on time unlike the other credit-card holder. But the penalty will only show in my account when I clear my dues and my fine.

```
(send my-card clear-outstanding-amount)
(send another-account show "abcd")
>4500.0
(send my-account show "amitabha")
>660.0
```

End of commentary.

I am posting on moodle, a tar-zipped files called `tutorial-problem-set-5-6.7.tgz` and `tutorial-problem-set-5-6.11.tgz`. It will unzip into:

- a directory called `model-implementation`
- a file called `test-model-implementation.rkt`
- a directory called `compiled`

with the usual meanings. `test-model-implementation.rkt` has enough test cases to show what you have to do.