MCN7105: Structure and Interpretation of Computer Programs
Academic Year: 2021/2022

Mid Semester Project

In this exercise we will build a data object for the representation of a 2D point. Consider the implementation below

```
(define (make-point x y)
   (cons x y))
(define (point-x p)
   (car p))
(define (point-y p)
   (cdr p))
(define (pretty-print p)
   (list 'x: (point-x p) 'y: (point-y p)))
```

The above code shows the implementation of a point object that can be created using the *make-point* function. The accessor operations are provided by the functions *point-x* and *point-y*. Building on the above implementation, define the following operations for a point object.

1. Define point setter operations *set-x!* and *set-y!* to change the value of x and y coordinates, respectively

2. Define a point operation *clone* that takes as argument a point object and returns its copy. The cloned point object should be independent of the original one i.e, the changes on the cloned point object should not affect the original one and vice versa.

3. Define a point operation *distance* that takes as argument two point objects and returns the distance between the two points.

4. Define a point operation *translate* that takes as argument a point object, *dx, dy* and moves the x and y point coordinates by dx and dy, respectively.

5. Define a point operation *point=?* that takes as argument two point objects and returns true if the two point objects are equal, false otherwise.

6. Up to now, we defined all the operations as top level operations. Of course a good design decision could be to have all the operations private to the point object. Consider the following *make-point-2D* implementation which is a variation of *make-point*:

```
(define (make-point-2D x y)
   (let ((self (cons x y)))
   (define (point-x)
      (car self))
   (define (point-y)
      (cdr self))
   (define (pretty-print)
      (list 'x: (point-x self) 'y: (point-y self)))
   ;your definions go here operations
   ;...
   (lambda (message . args)
   (case message
      ((point-x) (point-x))
      ((point-y) (point-y))
   ;..
   ;other operations go here
   (else (error "UKNOWN MESSAGE"))))))
```

   Complete the above implementation of *make-point-2D* such that a point object created using *make-point-2D* responds to all the operations in 1-5. e.g.,

```
(define p1 (make-point-2D 4 2))
(p1 'point-x) => 4
(define p2 (p1 'clone))
(p1 'point=? p2) => #t
```

   Note that now some operations such as the *clone* do not require a point object to be passed as argument.

7. Re-implement the point *object* using a vector data structure instead of the pair structure.

The End