region.lisp

```
(defgeneric scale (region scalar)
  (:documentation "scale of REGION: all coordinates multiplied by SCALAR"))
(defgeneric move (region vector)
  (:documentation "translation of REGION with vector VECTOR"))
(defgeneric rotate (region angle)
  (:documentation "rotation  of vector VECTOR"))
(defgeneric area (region)
  (:documentation "area of REGION"))
(defgeneric perimeter (region)
  (:documentation "area of REGION"))

;;; Regions
(defclass region () ())

(defgeneric compute-area (region)
  (:documentation
   "effectively compute area of REGION"))

(defmethod area ((region region))
  (compute-area region))

;;; Polygons
(defclass polygon (region)
  ((corners :initarg :corners :accessor corners)))

(defun make-polygon (&rest corners)
  (make-instance 'polygon :corners corners))

(defmethod print-object ((polygon polygon) stream)
  (call-next-method)
  (format stream "(corners: ~A)" (corners polygon)))

(defmethod scale ((region polygon) scalar)
  (setf (corners region)
        (mapcar (lambda (corner) (* corner scalar))
                (corners region))))

(defmethod move ((region polygon) vector)
  (setf (corners region)
        (mapcar (lambda (corner) (+ corner vector))
                (corners region))))

(defmethod rotate ((polygon polygon) angle)
  (let ((z (exp (* #c(0 1) angle))))
    (setf (corners polygon)
          (mapcar (lambda (point) (* point z))
                  (corners polygon)))))

(defun rotate-left (l)
  (if (null (cdr l))
      l
      (append (cdr l) (list (first l)))))

(defun rotate-right (l)
  (append (last l) (butlast l)))

;;; aire = 1/2 abs(sum_i=1^n X_i (Y_{i+1} - Y_{i-1}
;;; Y_{n+1}=Y_1 Y_0=Y_n

(defun area-of-polygon (corners)
  (let ((sum
         (loop
          with xs = (mapcar #'realpart corners)
          with ys = (mapcar #'imagpart corners)
          for x in xs
```

region.lisp

```
          for ysuiv in (rotate-right ys)
          for yprec in (rotate-left ys)
          sum (* x (- ysuiv yprec)))))
    (/ (abs sum) 2)))

(defmethod compute-area ((polygon polygon))
  (area-of-polygon (corners polygon)))

(defun distance (p1 p2)
  (let* ((v (- p1 p2))
         (x (realpart v))
         (y (imagpart v)))
    (sqrt (+ (* x x) (* y y)))))

(defmethod perimeter ((polygon polygon))
  (loop
     for corner in (corners polygon)
     for next-corner in (rotate-left (corners polygon))
     sum (distance corner next-corner)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Circles
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defclass circle (region)
  ((radius :initform 0 :accessor radius :initarg :radius)
   (center :initform 0 :accessor center :initarg :center)))

(defun make-circle (radius &optional (center #C(0 0)))
  (make-instance 'circle :center center :radius radius))

(defmethod print-object ((circle circle) stream)
  (call-next-method)
  (format stream "(Center:~A, Radius:~A)" (center circle) (radius circle)))

(defmethod scale ((circle circle) scalar)
  (setf (center circle) (* scalar (center circle))
        (radius circle) (* scalar (radius circle))))

(defmethod move ((circle circle) vector)
  (setf (center circle) (+ (center circle) vector)))

(defmethod rotate ((circle circle) angle)
  (setf (center circle) (* (exp (* #c(0 1) angle)) (center circle))))

(defmethod perimeter ((circle circle))
  (* pi (radius circle)))
```