MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001 Structure and Interpretation of Computer Programs
Spring, 2007

Recitation 6, Wed. February 28

Data Abstraction Problems Dr. Kimberle Koile

Here is an abstraction for a vector, which represents a point (x,y) in the plane.

* (make-vect x y) constructs a vector
* (get-x v) accesses a vector's x coordinate
* (get-y v) accesses a vector's y coordinate

1. What is the contract for the vector abstraction?

(get-x (make-vect x y)) => x

(get-y (make-vect x y)) => y

2. What is the type of each procedure?

make-vect: number, number → vector

get-x: vector → number

get-y: vector → number

3. Implement the abstraction when a vector is represented by a *pair*.

```
(define (make-vect x y) (cons x y))
(define (get-x v) (car v))
(define (get-y v) (cdr v))
```

or, equivalently:

```
(define make-vect cons)
(define get-x car)
(define get-y cdr)
```

4. Implement the abstraction when a vector is represented by a *list*.

```
(define (make-vect x y) (list x y))
(define (get-x v) (car v))
(define (get-y v) (car (cdr v)))
```

or, equivalently:

```
(define make-vect list)
(define get-x car)
(define get-y cadr)
```

5. Implement the abstraction when a vector is represented by a procedure. (Hint: Write a procedure that takes one argument.)

Here's one way to do it:

```
(define (make-vect x y)
  (lambda (coord)
    (if (= coord 0) x y)))

(define (get-x v) (v 0))
(define (get-y v) (v 1))
```

6. Using the vector abstraction, write the following operations on vectors.

(a) (+vect v1 v2) adds two vectors v1 and v2 using vector addition

```
(define (+vect v1 v2)
 (make-vect (+ (get-x v1) (get-x v2))
         (+ (get-y v1) (get-y v2))))
```

(b) (scale-vect v k) multiplies vector v by the scalar value k

```
(define (scale-vect v k)
 (make-vect (* (get-x v1) k)
         (* (get-y v1) k)))
```

(c) (mag v) computes the length of a vector.

```
(define (mag v)
 (sqrt (+ (square (get-x v)) (square (get-y v)))))
```

(d (=vect? v1 v2) returns true if v1 is the same point as v2

```
(define (=vect? v1 v2)
  (and (= (get-x v1) (get-x v2))
       (= (get-y v1) (get-y v2))))
```

7. Define another abstraction, curve, to represent a sequence of line segments whose start and end points are vectors.

* (make-curve v) constructs an empty curve, i.e. having no line segments and start point v.
* (extend-curve v c) constructs a curve by inserting a new point v at the start of another curve c
* (start-point c) returns a curve's start point.
* (rest-of-curve c) removes a curve's first line segment and returns the rest of it.
* (empty-curve? c) returns true if a curve has no line segments.

(a) What is the contract for the curve abstraction?

```
(start-point (make-curve v)))  =>   v
(start-point (extend-curve v c))  =>  v
(rest-of-curve (extend-curve v c))  =>  c
(empty-curve? (make-curve v)  =>  #t
(empty-curve? (extend-curve v c)  =>  #f
```

(b) Implement the abstraction when a curve is represented as a *list of points*.

```
(define (make-curve v) (list v))
(define (extend-curve v c) (cons v c))
(define (start-point c) (car c))
(define (rest-of-curve c) (cdr c))
(define (empty-curve? c) (null? (cdr c)))
```

(c) Implement the abstraction when a curve is represented by a *start point followed by a list of difference vectors* between each subsequent pair of points in the curve. (You can use the vector operations defined in previous problems. Also assume that you have the procedure –vect, which subtracts two vectors.)

```
(define (make-curve v) (list v))
(define (extend-curve v c)
      (cons v (cons (-vect (car c) v) (cdr c))))   ;; add new difference vector between new and old start pts
(define (start-point c) (car c))
(define (rest-of-curve c)
      (cons (+vect (car c) (cadr c)) (cddr c)))  ;; create new start point from old start and first difference
(define (empty-curve? c) (null? (cdr c)))
```

Look carefully at extend-curve and rest-of-curve, which are the only functions that are different from the previous answer. Extend-curve replaces the old curve's start point (found by (car c)) with its difference vector from the new start point (v) before cons'ing on the new start point. But why isn't rest-of-curve just the same as cdr, as it was in the previous problem?

10. Using the curve abstraction and vector operations, define the following operations on curves. (The procedures below should be implementation independent, so be sure to use the above procedures rather than car and cdr!)

(a) (translate c v) translates every point in a curve by vector v.

```
(define (translate c v)
  (if (empty-curve? c)
      (make-curve (+vect (start-point c) v))
      (extend-curve (+vect (start-point c) v))
                    (translate (rest-of-curve c) v))))
```

(b) (scale c k) scales every point in a curve by a scalar value k.

```
(define (scale c k)
  (if (empty-curve? c)
      (make-curve (scale-vect (start-point c) k))
      (extend-curve (scale-vect (start-point c) k))
                    (scale-vect (rest-of-curve c) k))))
```

(c) (perimeter c) computes the sum of the lengths of the line segments in a curve.

```
(define (perimeter c)
  (if (empty-curve? c)
      0
      (let ((p1 (start-point c))
            (p2 (start-point (rest-of-curve c))))
        (+ (mag (-vect p1 p2))
           (perimeter (rest-of-curve c))))))
```

(d) (closed? c) tests whether the curve's start point is the same as its end point.

Here are two solutions.

```
(define (closed? c)
  (define (end-point c)
    (if (empty-curve? c)
        (start-point c)
        (end-point (rest-of-curve c))))
  (=vect? (start-point c) (end-point c)))

(define (closed? c)
  (define (ends-with? c v)
    (if (empty-curve? c)
        (=vect? (start-point c) v)
        (ends-width? (rest-of-curve c) v)))
  (ends-with? c (start-point c)))
```