EECS 111 Final
March 16, 2010

Don't panic! *Read each question through. If any part confuses you, come up and ask me privately. Watch your time. Don't spend forever on any one question. Write cleanly. If you need to make big changes, X out the current code, write "see back" and write your new version on the back, with the number of the question.*

Important: Read questions in order. Many use functions defined in previous questions. Just assume you have working answers, whether you do or not.

1. (5 pts) `(vector-pos item vector)` is supposed to return the 0-based leftmost position of *item* in *vector*, if any, else -1. Fix the broken code below to do this.

```
(check-expect (vector-pos 'c (vector 'a 'b 'c)) 2)
(check-expect (vector-pos 'd (vector 'a 'b 'c)) -1)
```

```
(define (vector-pos x v)
  (vector-search x v (- (length v) 1)))

(define (vector-search x v n)
  (cond
    (eq? x v[n]) n
    else (vector-pos x v (- n 1))))
```

Comment [CKR1]: vector-length

Comment [CKR2]: right to left won't get leftmost item, must start at 0

Comment [CKR3]: missing base case ((>= n (vector-length v)) -1)

Comment [CKR4]: (vector-ref v n)

Comment [CKR5]: vector-search

Comment [CKR6]: +

Comment [CKR7]: missing parens around branches

2. (5 pts) Define `(for-n fn n)` to call `(fn 0)`, `(fn 1)`, …, `(fn (- n 1))` in that order. `for-n` calls *fn* for effect only. `for-n` should return `(void)`.

```
(define (for-n fn n)
  (for-n-iter fn 0 n))

(define (for-n-iter fn i n)
  (if (< i n)
      (begin (fn i) (for-n-iter fn (+ i 1) n))
      (void)))
```

3) (5 pts) Define `(copy-vector dest src start n)` to copy the first *n* elements from the vector *src* into the vector *dest*, starting at location *start* in *dest*. `copy-vector` should return the `dest` vector. Hint: use `for-n`.

```
(check-expect (copy-vector (vector 1 2 3 4 5) (vector 6 7 8) 1 2)
              (vector 1 6 7 4 5))
```

```
(define (copy-vector dest src offset n)
  (for-n (lambda (i)
           (vector-set! dest (+ offset i)
                        (vector-ref src i)))
         n)
  dest)
```

4. (8 pts) Define `(append! lst1 lst2)` to return the result of appending `lst2` to the end of `lst1`. Except when `lst1` is empty, `append!` should destructively modify the end of `lst1` to point to `lst2`.

```
(check-expect (append! empty (list 1 2 3)) (list 1 2 3))
(check-expect (append! (list 1 2 3) (list 4 5)) (list 1 2 3 4 5))
(check-expect
  (let ((lst (list 1 2 3))) (append! lst (list 4 5)) lst)
  (list 1 2 3 4 5))
```

```
(define (append! lst1 lst2)
  (if (empty? lst1) lst2
      (begin (set-cdr! (last-pair lst1) lst2)
             lst1)))

(define (last-pair lst)
  (if (or (empty? lst) (empty? (cdr lst)))
      lst
      (last-pair (cdr lst))))
```

Comment [CKR8]: can't set car or cdr of empty

Comment [CKR9]: needed to return right value

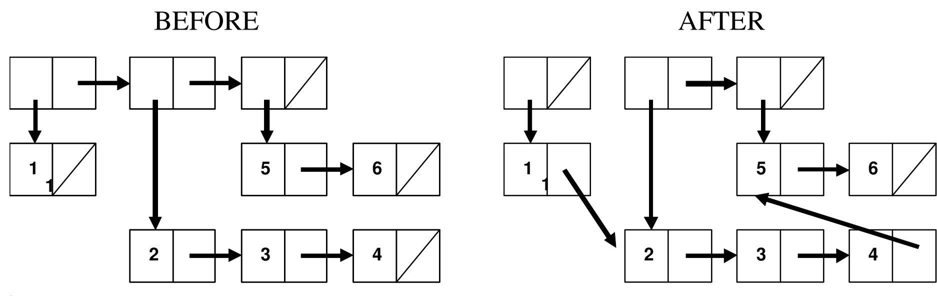
Comment [CKR10]: this code is straight from SICP and shown in class

5. Define (stitch! lsts) to take a list of sublists (possibly empty) and change it to a list of one list that is all the sublists destructively appended together.

```
(check-expect (stitch! empty) empty)
(check-expect (stitch! (list empty empty empty)) (list empty))
(check-expect (stitch! (list (list 1 2))) (list (list 1 2)))
(check-expect (stitch! (list (list 1) (list 2 3 4) (list 5 6)))
              (list (list 1 2 3 4 5 6)))
(check-expect
 (let ((lst (list empty (list 1 2) empty (list 3 4 5))))
   (stitch! lst)
   lst)
 (list (list 1 2 3 4 5)))
```

a) (4 pts) Draw two box and arrow diagrams, showing the fourth test case, before and after calling stitch!. Lay out the same boxes in both and change only the arrows.

.



`

b) (8 pts) Define stitch! Hint: use append!.

```
(define (stitch! lsts)
  (if (empty? lsts) lsts
      (begin
        (set-car! lsts (stitch-helper lsts))
        (set-cdr! lsts empty)
        lsts)))

(define (stitch-helper lsts)
  (if (empty? (cdr lsts))
      (car lsts)
      (append! (car lsts)
               (stitch-helper (cdr lsts)))))
```

Comment [CKR11]: needed to remove initial empty lists from car

Comment [CKR12]: needed to get rid of pointer lists

Comment [CKR13]: needed to return correct value

Comment [CKR14]: we know there's at least one list

6. (8 pts) In an object hierarchy, you define **classes** of objects, like UFO's and asteroids, with **parent classes**, e.g., "a UFO is a moving object." Each class also has a vector of variable names, e.g., **dx** and **dy** for the velocity of a moving object. For example, the following defines some classes for a possible game:

- a class **GameObject** with no parent and variables **x, y** for location
- a class **Mover** with parent **GameObject**, and variables **dx, dy** for velocity
- a class **UFO** with parent **Mover** and variable **torpedos** for how many photon torpedos it has

Note: `#(x y z …)` in Scheme creates a vector constant.

```
(define game-object (make-class false #(x y)))
(define mover (make-class game-object #(dx dy)))
(define ufo (make-class mover #(torpedos)))

(check-expect (class-parent ufo) mover)
(check-expect (class-variables mover) #(dx dy))
```

Define a Scheme `class` structure to make the above work. Then define the function `(superclass? class1 class2)` to return true if *class1* is *class2*, or a parent of *class2*, or a parent of a parent of *class2*, etc.

```
(check-expect (superclass? ufo ufo) true)
(check-expect (superclass? mover ufo) true)
(check-expect (superclass? game-object ufo) true)
(check-expect (superclass? ufo mover) false)
```

```
(define-struct class (parent variables))

(define (superclass? class1 class2)
  (and (not (false? class2))
       (or (eq? class1 class2)
           (superclass? class1
                        (class-parent class2)))))
```

Comment [CKR15]: more general than testing (false? (class-parent class2)) later

Comment [CKR16]: or equal? but not symbol-equal? or =

7. (5 pts) Assume the structure from #6. Define `(var-count class)` to return the number of variables relevant to *class*. That includes the variables in *class*, plus all those in its superclasses. E.g., the `ufo` class has 5 relevant variables, in this order: `torpedos, dx, dy, x, y`.

```
(check-expect (var-count ufo) 5)
(check-expect (var-count game-object) 2)
```

```
(define (var-count class)
  (if (false? class) 0
      (+ (vector-length (class-variables class))
         (var-count (class-parent class)))))
```

Comment [CKR17]: simpler and more general base case than (false (class-parent class))

8. (8 pts) Define `(class-vars class)` to construct a vector of the variables relevant to *class*, starting with *class*'s variables, then its parent's, then the parent's parent, and so on. Hint: use `var-count` and `copy-vector`.

```
(check-expect (class-vars game-object) #(x y))
(check-expect (class-vars ufo) #(torpedos dx dy x y))
```

```
(define (class-vars class)
  (collect-vars class
    (make-vector (var-count class) false)
    0))

(define (collect-vars class vector offset)
  (if (false? class) vector
      (let* ((vars (class-variables class))
             (len (vector-length vars)))
        (collect-vars (class-parent class)
           (copy-vector vector vars offset len)
           (+ offset len)))))
```

Comment [CKR18]: new vector critical – don't modify class-variables, which is too short to copy into anyway

Comment [CKR19]: make-vector + copy-vector much cheaper than multiple vector-appends

9. (4 pts) Define a structure for an **instance** of *`class`* that contains *`class`*, a vector of all the variables relevant to *`class`*, and a vector of values for those variables. Define `(new-instance` *`class values`*`)` to create such a structure for *`class`*, with the given values. Hint: use `class-vars`.

```
(define ufo-1 (new-instance ufo (vector 8 -1 1 10 30)))
(define sun-1 (new-instance game-object (vector 200 300)))

(check-expect (instance-class ufo-1) ufo)
(check-expect (instance-variables ufo-1) #(torpedos dx dy x y))
(check-expect (instance-variables sun-1) #(x y))
(check-expect (instance-values ufo-1) #(8 -1 1 10 30))
```

```
(define-struct instance (class variables values))

(define (new-instance class values)
  (make-instance class (class-vars class) values))
```

10. (6 pts) Define `(var-get` *`instance var-name`*`)` and `(var-set!` *`instance var-name value`*`)` to get and set the appropriate instance value for *`var-name`*. Hint: use `vector-pos`.

```
(check-expect (var-get ufo-1 'torpedos) 8)
(check-expect (var-get ufo-1 'x) 10)
(check-expect
  (let ((m (new-instance mover (vector 1 -1 10 20))))
    (var-set! m 'x 50)
    (instance-values m))
  #(1 -1 50 20))
```

```
(define (var-get instance var)
  (vector-ref (instance-values instance)
    (vector-pos var
                (instance-variables instance))))

(define (var-set! instance var value)
  (vector-set! (instance-values instance)
     (vector-pos var
                 (instance-variables instance))
     value))
```