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 (vector-ref v n)) n]
         [else (vector-pos x v (- n 1))] )
   )

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 \((\text{copy-vector } \textit{dest} \textit{src} \textit{start} \textit{n})\) to copy the first \(n\) elements from the vector \(\textit{src}\) into the vector \(\textit{dest}\), starting at location \(\textit{start}\) in \(\textit{dest}\). \(\text{copy-vector}\) should return the \(\textit{dest}\) vector. Hint: use \(\text{for-n}\).

\[
(\text{check-expect (copy-vector (vector 1 2 3 4 5) (vector 6 7 8) 1 2)
(vector 1 6 7 4 5))}
\]

\[
(\text{define (copy-vector dest src offset n)}
(\text{for-n (lambda (i)}
  (vector-set! dest (+ offset i)
    (vector-ref src i)))
  n)
\text{dest})
\]

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

\[
(\text{check-expect (append! empty (list 1 2 3)) (list 1 2 3))}
(\text{check-expect (append! (list 1 2 3) (list 4 5)) (list 1 2 3 4 5))}
(\text{check-expect}
  (let ((lst (list 1 2 3))) (append! lst (list 4 5)) lst)
  (list 1 2 3 4 5))
\]

\[
(\text{define (append! lst1 lst2)}
  (if (empty? lst1) lst2
    (begin (set-cdr! (last-pair lst1) lst2)
      lst))
\]

\[
(\text{define (last-pair lst)}
  (if (or (empty? lst) (empty? (cdr lst)))
    lst
    (last-pair (cdr lst)))
\]

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 \( \texttt{stitch! lsts} \) 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.

\[
\begin{align*}
(\text{check-expect} & \ (\texttt{stitch! empty}) \ \text{empty}) \\
(\text{check-expect} & \ (\texttt{stitch! (list empty empty empty)}) \ (\text{list empty})) \\
(\text{check-expect} & \ (\texttt{stitch! (list (list 1 2))) (list (list 1 2))))) \\
(\text{check-expect} & \ (\texttt{stitch! (list (list 1) (list 2 3 4) (list 5 6))}) \\
(\text{list (list 1 2 3 4 5 6)))) \\
(\text{check-expect} & \ \\
(\text{let} & \ ((\texttt{lst (list empty (list 1 2) empty (list 3 4 5))})) \\
(\texttt{stitch! lst}) \\
(\text{list (list 1 2 3 4 5)))) \\
\end{align*}
\]

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

\[
\begin{array}{cc}
\text{BEFORE} & \text{AFTER} \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & \rightarrow & \text{\ } \\
2 & \rightarrow & 3 \\
\rightarrow & & 4 \\
\rightarrow & & 5 \rightarrow 6 \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & \rightarrow \text{\ } \\
2 & \rightarrow 3 & \rightarrow 4 \\
\rightarrow & 5 & \rightarrow 6 \\
\end{array}
\]

b) (8 pts) Define \( \texttt{stitch!} \). Hint: use \( \texttt{append!} \).

\[
\begin{align*}
(\text{define} & \ (\texttt{stitch! lsts}) \\
(\text{if} & \ (\text{empty? lsts}) \ \texttt{lsts}) \\
(\text{begin} & \ ( \text{set-car! lsts (stitch-helper lsts)}) \\
& \ (\texttt{set-cdr! lsts empty}) \\
& \ \texttt{lsts}))) \\
(\text{define} & \ (\texttt{stitch-helper lsts}) \\
(\text{if} & \ (\text{empty? (cdr lsts)}) \\
(\texttt{car lsts}) \\
(\text{append!} & \ (\texttt{car lsts}) \\
& \ (\texttt{stitch-helper (cdr lsts)}))))
\end{align*}
\]
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.

(define-struct class (parent variables))

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

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

\[
\begin{align*}
(\text{check-expect} \ (\text{var-count} \ \text{ufo}) & \ 5) \\
(\text{check-expect} \ (\text{var-count} \ \text{game-object}) & \ 2) \\
\end{align*}
\]

\[
\text{(define} \ (\text{var-count} \ \text{class}) \\
(\text{if} \ (\text{false?} \ \text{class}) \ 0 \\
\quad (+ \ (\text{vector-length} \ (\text{class-variables} \ \text{class}))) \\
\quad (\text{var-count} \ (\text{class-parent} \ \text{class}))))
\]

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

\[
\begin{align*}
(\text{check-expect} \ (\text{class-vars} \ \text{game-object}) & \ (x \ y)) \\
(\text{check-expect} \ (\text{class-vars} \ \text{ufo}) & \ (\text{torpedos \ dx \ dy \ x \ y})) \\
\end{align*}
\]

\[
\text{(define} \ (\text{class-vars} \ \text{class}) \\
(\text{collect-vars} \ \text{class} \\
[\text{make-vector} \ (\text{var-count} \ \text{class}) \ \text{false}] \\
0))
\]

\[
\text{(define} \ (\text{collect-vars} \ \text{class \ vector \ offset}) \\
(\text{if} \ (\text{false?} \ \text{class}) \ \text{vector} \\
\quad (\text{let*} \ ((\text{vars} \ (\text{class-variables} \ \text{class}))) \\
\quad \quad (\text{len} \ (\text{vector-length} \ \text{vars}))) \\
\quad (\text{collect-vars} \ (\text{class-parent} \ \text{class}) \\
\quad \quad (\text{copy-vector} \ \text{vector} \ \text{vars} \ \text{offset} \ \text{len}) \\
\quad \quad (+ \ \text{offset} \ \text{len}))))
\]
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-values ufo-1) #(8 -1 1 10 30))
```

```
(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-values ufo-1) #(8 -1 1 10 30))
```

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.

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

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

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

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

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

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