Solutions for Mid-Term Quiz

September 28, 2011

Time allowed: 1 hour 45 minutes

Matriculation No: [ ] [ ] [ ] [ ] [ ] [ ]

Instructions (please read carefully):

1. Write down your matriculation number on the question paper. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is an open-sheet quiz. You are allowed to bring one A4 sheet of notes (written on both sides).
3. This paper comprises FOUR (4) questions and SIXTEEN (16) pages. The time allowed for solving this quiz is 1 hour 45 minutes.
4. The maximum score of this quiz is 100 marks. The weight of each question is given in square brackets beside the question number.
5. All questions must be answered correctly for the maximum score to be attained.
6. All questions must be answered in the space provided in the answer sheet; no extra sheets will be accepted as answers.
7. The back-sides of the sheets and the pages marked “scratch paper” in the question set may be used as scratch paper.
8. You are allowed to un-staple the sheets while you solve the questions. Please make sure you staple them back in the right order at the end of the quiz.
9. You are allowed to use pencils, ball-pens or fountain pens, as you like (no red color, please).

GOOD LUCK!

<table>
<thead>
<tr>
<th>Question</th>
<th>Marks</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Question 1: Contract Fever [32 marks]

In class, we discussed specifications and it was said that a specification can have any implementation as long as the implementation satisfies the contract. The contract for cons is as follows:

- \((\text{car} \ (\text{cons} \ x \ y))\) returns \(x\).
- \((\text{cdr} \ (\text{cons} \ x \ y))\) returns \(y\).

In particular, we discussed in lecture the following implementation of cons, car and cdr that satisfies the contract:

\[
\begin{align*}
(\text{define} \ (\text{cons} \ x \ y) \\
(\lambda (m) \ (m \ x \ y)))
\end{align*}
\]

\[
\begin{align*}
(\text{define} \ (\text{car} \ z) \\
(z \ (\lambda (p \ q) \ p)))
\end{align*}
\]

\[
\begin{align*}
(\text{define} \ (\text{cdr} \ z) \\
(z \ (\lambda (p \ q) \ q)))
\end{align*}
\]

In this question, we provide you with different implementations of cons (which we called new-cons to avoid possible naming conflicts). Your task is to suggest possible implementations of the corresponding new-car and new-cdr in each case to satisfy the contract:

- \((\text{new-car} \ (\text{new-cons} \ x \ y))\) returns \(x\).
- \((\text{new-cdr} \ (\text{new-cons} \ x \ y))\) returns \(y\).

A. \((\text{define} \ (\text{new-cons} \ x \ y) \\
(\text{list} \ 	ext{'*cool*} \ x \ (\text{list} \ y)))\)

\[
\begin{align*}
(\text{define} \ (\text{new-car} \ z) \\
<T1>)
\end{align*}
\]

\[
\begin{align*}
(\text{define} \ (\text{new-cdr} \ z) \\
<T2>)
\end{align*}
\]

T1: This question tests if the student understands how to navigate cons structures.

\((\text{cadr} \ z)\)

T2: \((\text{caaddr} \ z)\)

[4 marks]
B. (define (new-cons x y)
   (lambda (p q)
      (if (q p)
         x
         y))))

(define (new-car z)
  <T3>)

(define (new-cdr z)
  <T4>)

T3: [4 marks]
This question tests if the student is able to wrap his/her head around the passing of lambdas.

  (z 'anything (lambda (x) #t))

T4: [4 marks]

  (z 'anything (lambda (x) #f))

C. (define (new-cons x y)
    (if x
        (list "cool" y x)
        (list "not-cool" x y)))

(define (new-car z)
  <T5>)

(define (new-cdr z)
  <T6>)

T5: [4 marks]
This question tests if the student understands the use of equals? to compare strings. "if x" is a red herring.

  (if (equal? "cool" (car z)) (caddr z) (cadr z))

T6: [4 marks]

  (if (equal? "cool" (car z)) (cadr z) (caddr z))
D.  (define (new-cons x y)
    (let* ((p x)
            (q p)
            (y q))
      (lambda (s t) (t (s x y) p)))))

(define (new-car z)
  <T7>)

(define (new-cdr z)
  <T8>)

This question tests if the student understands the use of let* and also lambdas (of course!). Students cannot assume that x and y are integers. 2 points will be taken off if solution assumes that x and y are integers.

(z (lambda (x y) x) (lambda (x y) x))

NO solution. The question tests if the student understands the concept of aliasing. Also, students need to learn that in life, not every question necessarily has a solution.

4
Question 2: Fun with Continued Fractions  [25 marks]

Consider the following set of continued fractions:

\[
\begin{align*}
    f_x(1) &= \frac{1}{x} \\
    f_x(2) &= \frac{1}{1 - \frac{1}{x}} \\
    f_x(3) &= \frac{1}{1 - \frac{1}{1 - \frac{1}{x}}} \\
    &\quad \vdots \\
    f_x(n) &= \frac{1}{1 - \frac{1}{\ldots - \frac{1}{x}}} 
\end{align*}
\]

For all sections in the question, you may assume \(n \geq 1\).

A. Please give an implementation for the procedure \(f\), that takes in two parameters \(n\) and \(x\) and computes \(f_x(n)\).  
[6 marks]

Sample execution:

\[
\begin{align*}
(f 1 2) &> 1/2 \\
(f 2 2) &> 2 \\
(f 3 2) &> -1
\end{align*}
\]

\[
\begin{align*}
\text{(define (f n x)} \\
&\quad (\text{if (= n 1)} \\
&\quad \quad (\text{(/ 1 x)} \\
&\quad \quad \quad (\text{(/ 1 (- 1
&\quad \quad \quad \quad (f (- n 1
&\quad \quad \quad \quad \quad x)))))}))}
\end{align*}
\]
B. Is the procedure you wrote in Part (A) above a recursive or iterative process? What is the order of growth in terms of time and space? [3 marks]

Procedure in Part A is a recursive process.

Time: $O(n)$

Space: $O(n)$

Now consider the following set of continued fractions:

\[
\begin{align*}
g_{x,y}(1) & = \frac{1}{x} \\
g_{x,y}(2) & = \frac{1}{y - \frac{1}{x}} \\
g_{x,y}(3) & = \frac{1}{y - \frac{1}{y - \frac{1}{x}}} \\
& \vdots \\
g_{x,y}(n) & = \frac{1}{y - \frac{1}{\ldots - \frac{1}{x}}} 
\end{align*}
\]

Obviously, $f_x(n) = g_{x,1}(n)$. The procedure $g$ computes $g_{x,y}(n)$ when given the parameters $n$, $x$ and $y$.

C. Given:

\[
\begin{align*}
\text{(define (fold-up op f n)}
& \quad \text{(define (iter result count)}
& \quad \quad \text{(if (> count n)}
& \quad \quad \quad \text{result)
& \quad \quad \quad \text{(iter (op result (f count))}
& \quad \quad \quad \quad \text{(+ count 1)))}
& \quad \text{)}
& \quad \text{)}
& \quad \text{(define (g n x y)}
& \quad \text{(fold-up <T9>}
& \quad \quad \text{<T10>}
& \quad \quad \quad \text{<T11>))}
\end{align*}
\]

Give a possible implementation for each of the terms $T9$, $T10$, and $T11$: [6 marks]

\[
\begin{align*}
T9: \quad \text{(lambda (p q) (/ 1 (- y p)))}
\end{align*}
\]

[2 marks]

Observe that $(\text{fold-up op f n}) \equiv$

\[
\text{(op ... (op (f 0) (f 1)) f 2) ... ).}
\]
### T10: 
2 marks

\[
\lambda (n) \left( \frac{1}{x} \right)
\]

### T11: 
2 marks

\[- n 1\]

### D. Given:

\[
\text{(define (accumulate op initial sequence)}
\]

\[
\text{if (null? sequence)}
\]

\[
\text{initial}
\]

\[
\text{(op (car sequence)}
\]

\[
\text{(accumulate op initial (cdr sequence)))}}
\]

\[
\text{(define (g n x y)}
\]

\[
\text{(accumulate <T12>}
\]

\[
\text{<T13>}
\]

\[
\text{(make-sequence n x y))}
\]

\[
\text{(define (make-sequence n x y)}
\]

\[
\text{<T14>)}
\]

Give a possible implementation for each of the terms T12, T13, and T14:  
10 marks

### T12: 
3 marks

\[
\lambda (p \ q) \left( \frac{1}{- \ p \ q} \right)
\]

Observe that \((\text{accumulate op initial (s1 s2 s3 ... s4)}) \equiv (\text{op s1 (op s2 (op s3 ... (op sn initial) ... )})\). 

### T13: 
3 marks

\[ \frac{1}{x} \]

### T14: 
4 marks

\[
\text{(if (= n 1)}
\]

\[
\text{'}()
\]

\[
\text{(cons y (make-sequence (- n 1) x y))})
\]
Question 3: Doubly-Linked Lists [18 marks]

In Scheme, the lists that are created with cons boxes are typically known as linked lists. Typically we traverse linked lists by cdring down a list in one direction. It is also possible to create lists that are called doubly-linked lists. For such lists, it is possible to traverse the list in both directions.

Doubly-linked lists consists of nodes (which are like cons boxes) that have three entries instead of two. One entry is a pointer to an object, while the other two are pointers to the previous and next nodes. The nodes at the beginning and end of a doubly-linked list would have null pointers as their previous and next pointers respectively. The following is a graphical representation of a doubly-linked list:

![Diagram of a doubly-linked list]

[Wishful Thinking] Suppose that the procedure make-node creates a node for a doubly-linked list that satisfies the following contract:

- (get-object (make-node object prev next)) returns object.
- (get-prev (make-node object prev next)) returns prev.
- (get-next (make-node object prev next)) returns next.

An empty doubly-linked list is represented with a null pointer.

A. Write a procedure dlist->list that takes in a doubly-linked list and returns a regular list with the elements in the same order. [6 marks]

```
(define (dlist->list dlst)
  (if (null? dlst)
      ()
      (cons (get-object dlst)
            (dlist->list (get-next dlst)))))
```
[More Wishful Thinking] Suppose you now have three more procedures set-object!, set-prev! and set-next! that when applied to a node n and a value v, will set the object pointer, previous pointer and next pointer to the value v respectively.

B. Write a procedure list->dlist that takes in a regular list and returns a doubly-linked list with the elements in the same order. The return value should be a reference to the first node of the doubly-linked list. [6 marks]

This is a test of whether students can figure out how to pass references around. Basically, we need to keep a reference to the first node of the doubly-linked list as we cdr down and create the list. If not, this question will teach them how it’s done.

```
(define (list->dlist lst)
  (define (helper lst prev)
    (if (null? lst)
      '()
      (let* ((next (list->dlist (cdr lst)))
              (current (make-node (car lst) prev next)))
        (if (not (null? next))
          (set-prev! next current))
        current))
  (helper lst '()))

It is possible to use a list to keep tracker of all the new nodes and then write a procedure to fix all the pointers:

(define (list->dlist lst)
  (define (fix-pointers lst prev)
    (if (null? lst)
      'done
      (let ((current (car lst)))
        (set-prev! current prev)
        (if (not (null? (cdr lst)))
          (set-next! current (cadr lst)))
        (fix-pointers (cdr lst) current)))))
  (if (null? lst)
    '()
    (let ((dlist (map (lambda (x) (make-node x () ()) lst)))
      (fix-pointers dlist '())
      (car dlist))))
```
C. Write a procedure reverse-dlist that takes in a doubly-linked list and returns a new doubly-linked list with the elements in the reversed order.  

Sometimes, there is a need to think out of the box. :-)

(define (reverse-dlist dlst)
  (list->dlist (reverse (dlist->list dlst)))))
Question 4: Fun with Word Puzzles (or Word Puzzle Horror) [25 marks]

In this problem, you will work on word puzzles like the following:

```
f e l p p a
u r a t e r
h e c a r i
e h o m e s
l t a e s i
p o t s e r
```

For these word puzzles, the goal is to find words hidden in the puzzle that can be read either horizontally or vertically in either direction. Some variants of the puzzle also allow for words in the diagonal direction, but for this problem, we shall keep it simple and allow words to read only either horizontally or vertically (but in both directions).

We will represent the puzzle with a list of lists of symbols as follows:

```
(define puzzle '(((f e l p p a)
                  (u r a t e r)
                  (h e c a r i)
                  (e h o m e s)
                  (l t a e s i)
                  (p o t s e r)))
```

For the rest of this problem, assume that an \( n \times n \) word puzzle is represented with a list of \( n \) lists, each consisting of \( n \) symbols.

A. Write a procedure `extract-columns` that takes in a puzzle and returns a list of all the columns in the puzzle from left to right. Ordering from up to down should be preserved.[6 marks]

Sample execution:

```
(extract-columns puzzle)
> ((f u h e l p) (e r e h t o) (l a c o a t) (p t a m e s) (p e r e s e) (a r i s i r))
```

```
(define (extract-columns puzzle)
  (if (null? (car puzzle))
      '()
      (cons (map car puzzle)
            (extract-columns (map cdr puzzle))))
```

B. Words are also represented with lists of symbols. For example, the list `(a p p l e)` is used to represent the word “apple.” Write a procedure `match-word?` that takes in a word (which is a list of symbols) and a row of symbols and returns `#t` if the word is a sub-sequence of the row of symbols, or `#f` otherwise. [7 marks]

Sample execution:

```scheme
(match-word? '(a p p l e) '(r a p p l e r e a p))
> #t

(match-word? '(p e a r) '(r a p p l e r e a p))
> #f
```
(define (match-word? word row)
  (define (exact-match? word row) ; match word with beginning of row
    (cond ((null? word) #t)
           ((not (eq? (car word) (car row))) #f)
           (else (exact-match? (cdr word) (cdr row))))

    (cond ((or (null? row) (< (length row) (length word))) #f)
          ((exact-match? word row) #t)
          (else (match-word? word (cdr row))))))

A more elegant solution (by a student):
(define (match-word? word row)
  (define (helper tword trow)
    (cond ((null? tword) #t)
          ((null? trow) #f)
          ((eq? (car tword) (car trow))
           (helper (cdr tword) (cdr trow)))
          (else (match-word? word (cdr row))))

  (helper word row))

C. A dictionary is a list of words. Write a procedure solve that takes in a puzzle and a dictionary and returns a list of all the words in the dictionary that can be found in the puzzle. [12 marks]

Sample execution:
(define dictionary1 '((a p p l e) (t r e e) (r a t e) (h o m e) (s i r) (h e r) (p e a r) (r o o t) (s e a t)))

(solve puzzle dictionary1)
> ((a p p l e) (r a t e) (h o m e) (s i r) (h e r) (s e a t))

(define dictionary2 '((g o o d) (b o y) (g i r l s) (l o v e) (c s 1 1 0 1 s)))

(solve puzzle dictionary2)
> ()

[Wishful Thinking] We generate all the rows and columns of the puzzle, both forward and reverse. Then for each word, we check whether we can find a match in any of the rows. QED.
(define (solve puzzle dictionary)
  (let ((rows (generate-all puzzle)))
    (filter (lambda (x) x) ; why is this?
      (map (lambda (word) (find-match? word rows))
           dictionary))))

(define (generate-all puzzle)
  (let ((columns (extract-columns puzzle)))
    (append puzzle
             (map reverse puzzle)
             columns
             (map reverse columns))))

(define (find-match? word rows) ; returns #f is not found, word otherwise
  (cond ((null? rows) #f)
        ((match-word? word (car rows)) word)
        (else (find-match? word (cdr rows))))))
Appendix

The following are some procedures that were introduced in class. For your reference, they are reproduced here.

(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ result (term a))))
  (iter a 0))

(define (fold op f n)
  (if (= n 1)
      f 1
      (op (f n)
          (fold op f (- n 1))))

(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))))

(define (enumerate-interval low high)
  (if (> low high)
      ()
      (cons low (enumerate-interval (+ low 1) high))))

(define (filter predicate sequence)
  (cond ((null? sequence) ())
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence))))

(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial
                      (cdr sequence)))))

The procedure even? is a Scheme primitive that returns #t if its argument is an even number, and #f otherwise.

The procedure odd? is a Scheme primitive that returns #t if its argument is an odd number, and #f otherwise.