CSE 1729 - Introduction to Principles of Programming — December 5, 2017

Lab 13 – Final Exam Practice

The tasks here are problems for you to practice. They vary a good deal in difficulty. These are not to be submitted.

Enjoy!

The solutions will be published before the final, but not long before.

Expressions in SCHEME

Write a SCHEME expression to evaluate:

1. the sum of 4, 8, 15, 16, 23 and 42

   Solution:

   ```
   (+ 4 8 15 16 23 42)
   ```

2. the product of 653,854,321 and 241,304,201

   Solution:

   ```
   (* 633854321 241304201)
   ```

3. $\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}$

   Solution:

   ```
   (/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5))))) (* 3 (- 6 2) (- 2 7)))
   ```

Functions

1. Simple Functions

   (a) Define a SCHEME function which takes a number as a parameter and returns the absolute value of that number.

   Solution:

```
; note: abs is function that does this in R5RS
(define (absolute x)
  (if (< x 0)
      (- x)
      x))
```

(b) Define a function to convert a temperature from Fahrenheit to Celsius, and another to convert in the other direction. The two formulas are $F = \frac{9}{5}C + 32$ and $C = \frac{5}{9}(F - 32)$.

Solution:

```
(define (fahr-to-cels f) (* (- f 32) (/ 5 9)))
(define (cels-to-fahr c) (+ (* (/ 9 5) c) 32))
```

(c) Define a SCHEME function named `discount` which takes a product's price and discount (as a percent) and returns the discounted price of the product.

Solution:

```
(define (discount price percent)
  (* price (- 1 percent)))
```

(d) Write a procedure to compute the tip you should leave at a restaurant. It should take the total bill as its argument and return the amount of the tip. It should tip by 20%, but it should know to round up so that the total amount of money you leave (tip plus original bill) is a whole number of dollars. (Use the `ceiling` procedure to round up.)

Solution:

```
(define (tip bill) (- (ceiling (* 1.2 bill)) bill))
```

(e) Most latex paints cover approx. 400 sq. ft. of area. Stains and varnishes cover 500 sq.ft. of area. Write a SCHEME function that takes the length and width (or height) of an area to be covered as well as if the area is to be painted or stained (a boolean parameter would be useful here) and returns the number of gallons required to finish one coat.

Solution:

```
(define (num-gallons l w p?)
  (let ((area (* l w)))
    (if p?
        (ceiling (/ area 400))
        (ceiling (/ area 500)))))
```

(f) Suppose you are contacted by a client who lives in a house where all of the ceilings are hemispheres; he would like a SCHEME function that takes the radius of a ceiling (in feet), plus whether it is to be painted or stained, and returns the number of gallons required to paint one coat on the ceiling. Feel free to use your function from part (e) as a helper.

Solution:

```
(define (num-gallons-ceiling r p?)
  (num-gallons (acos -1) (* 2 r r) p?))
```

2. Recursion

(a) (Towers of Hanoi) The Towers of Hanoi is a famous puzzle with a recursive solution. There are many versions of the legend. One version of the description tells of a temple where the monks spend their days transferring large disks of different sizes between three pegs (each forming a tower). Disks can only be moved according to the following rules:

1. Can only move one disk at a time
2. Disks must be properly stacked, disks may only be placed on larger disks
3. All disks but one must be on one of the three pegs

The recursive solution to this problem labels the three pegs as the source, destination and temporary pegs. The recursive decomposition follows the following form; a stack of n disks on the source peg can be moved to the destination peg by moving the top $n-1$ disks to the temporary peg, moving the largest disk to the destination peg and then moving the $n-1$ smallest disks to the destination peg. This gives us the following algorithm for a function that takes four parameters. The pegs labeled *source*, *temp* and *dest* as well as the number of disks to be moved.

`(define (towers-of-hanoi n source temp dest)`

1. `(towers-of-hanoi n-1 source dest temp)`
2. `move disk n from source to dest`
3. `(towers-of-hanoi n-1 temp source dest))`

Define a SCHEME function that lists steps one can follow to solve the Towers of Hanoi puzzle. You can use the SCHEME function `display` to print a message indicating which disk to move and which pegs to move it from and to (step (b) in the algorithm above). Parameters representing pegs can accept integers identifying each of the pegs (e.g. 1 2 and 3).

Solution:

```
(define (towers-of-hanoi n origin dest temp)
  (cond ((not (= n 0))
         (begin (towers-of-hanoi (- n 1) origin temp dest)
                (display "Move disk ")
                (display n)
                (display " to peg ")
                (display dest)
```

```
                (display #\newline)
                (towers-of-hanoi (- n 1) temp dest origin)))))
(towers-of-hanoi 4 'A 'B 'C)
```

3. Tail Recursion

   (a) Write 2 SCHEME functions that takes a list of numbers as a parameter and returns the number of non-negative numbers. One should be tail-reursive, the other should not be.

Solution:

```
;not tail recursive
(define (count-nns lst)
  (cond ((null? lst) 0)
        ((>= (car lst) 0)
         (+ 1 (count-nns (cdr lst))))
        (else
         (count-nns (cdr lst)))))

;tail recursive (aka iterative)
(define (count-nns-it lst)
  (define (count-nns-iter lst count)
    (cond ((null? lst) count)
          ((>= (car lst) 0)
           (count-nns-iter (cdr lst) (+ count 1)))
          (else
           (count-nns-iter (cdr lst) count))))
  (count-nns-iter lst 0))
```

   (b) Write a SCHEME function that takes two integers as parameters and returns their greatest common divisor (GCD) using Euclid's Algorithm. From Wikipedia: "The GCD of two numbers is the largest number that divides both of them without leaving a remainder. The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the smaller number is subtracted from the larger number. For example, 21 is the GCD of 252 and 105 ($252 = 21 \times 12; 105 = 21 \times 5$); since 252 - 105 = 147, the GCD of 147 and 105 is also 21. Since the larger of the two numbers is reduced, repeating this process gives successively smaller numbers until one of them is zero. When that occurs, the GCD is the remaining nonzero number."

Solution:

```
(define (GCD a b)
  (cond ((= a 0) b)
        ((= b 0) a)
```

```
        ((< a b) (GCD a (- b a)))
        (else (GCD (- a b) b))))
```

4. Higher-Order Functions

   (a) (SICP Exercise 1.43) If f is a numerical function and n is a positive integer, then we can form the n^{th} repeated application of f, which is defined to be the function whose value at x is $f(f(...(f(x))...))$. For example, if f is the function $x \mapsto x+1$, then the n^{th} repeated application of f is the function $x \mapsto x+n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the $2^{n^{th}}$ power. Write a procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f. Your procedure should be able to be used as follows:

```
((repeated square 2) 5)
625
```

   Hint: You may find it convenient to use compose from exercise 1.42.

Solution:

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define (repeated f n)
  (if (= n 1)
      f
      (compose f (repeated f (- n 1)))))
```

   (b) (SICP Exercise 1.44) The idea of smoothing a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x-dx)$, $f(x)$, and $f(x+dx)$. Write a procedure smooth that takes as inputs a procedure that computes f and a number dx and returns a procedure that computes the smoothed f.

Solution:

```
(define (smooth f dx)
  (lambda (x) (/ (+ (f (- x dx))
                    (f x)
                    (f (+ x dx)))
                 3)))
```

List Processing

1. Pairs

   (a) Define a function that takes two pairs, representing two points, as parameters and returns a pair consisting of the slope and y-intercept of the line intersecting those two points.

   Solution:

```
(define (line-equation a b)
  (let ((m (/ (- (cdr a) (cdr b)) (- (car a) (car b)))))
    (cons m
          (- (cdr a) (* m (car a))))))
```

   (b) (Unusual Canceling) The fraction $\frac{64}{16}$ exhibits the unusual property that its reduced value of 4 may be obtained by "canceling" the 6 in the numerator with the 6 in the denominator. Define a SCHEME function which takes a fraction represented as a pair and returns a boolean value indicating whether the value of the fraction remains unchanged after this unusual canceling.

   If xy is the two digit number (e.g. if $n = 64$ then $x = 6$ and $y = 4$) then

   1. $n = 10x + y$
   2. $x =$ `(floor (/ n 10))`
   3. $b =$ `(modulo n 10)`

   Solution:

```
(define (unusual-canceling num denom)
  (let ((x-num (floor (/ num 10)))
        (y-num (modulo num 10))
        (x-denom (floor (/ denom 10)))
        (y-denom (modulo denom 10)))

    (if (= x-num y-denom)
        (= (/ num denom) (/ y-num x-denom))
        #f)))
```

2. Lists

   (a) Write a function that computes the largest distance between any two adjacent elements of a list.

   Solution:

```
(define (max-pair-dist lst)
  (define (max-pair-dist-aux lst max)
    (if (null? (cdr lst))
        max
```

```
        (if (> (absolute (- (car lst) (cadr lst))) max)
            (max-pair-dist-aux (cdr lst)
                               (absolute (- (car lst)
                                            (cadr lst))))
            (max-pair-dist-aux (cdr lst) max))))
  (max-pair-dist-aux lst 0))
```

(b) Write a function that, given a list $(a_1 \dots a_k)$, returns the list $(b_1 \dots b_{k-1})$ so that $b_i = a_{i+1} - a_i$.

Solution:

```
(define (difference-list lst)
  (if (null? (cdr lst))
      '()
      (cons (- (cadr lst) (car lst))
        (difference-list (cdr lst)))))
```

(c) Write a function which takes a list of numbers as parameters and returns a pair representing the range of the numbers in the list. That is, the first in the pair returned should be the minimum element of the list and the second should be the maximum.

Solution:

```
(define (list-range lst)
  (define (list-range-aux lst min max)
    (if (null? lst)
        (cons min max)
        (let ((new-min (if (< min (car lst)) min (car lst)))
              (new-max (if (> max (car lst)) max (car lst))))
          (list-range-aux (cdr lst) new-min new-max))))
  (list-range-aux (cdr lst) (car lst) (car lst)))
```

(d) Define a SCHEME function which uses the solution to the Pairs question b to produce a list of pairs representing all of the fractions which exhibit the unusual canceling property. Note: you may limit yourself to fractions which have only two digits in their numerators and denominators.

Solution:

```
(define (list-unusual-canceling)
  (define (list-unusual-canceling-aux num denom)
    (cond ((> num 99) '())
          ((> denom 99)
```

```
          (list-unusual-canceling-aux (+ num 1) 10))
         ((unusual-canceling num denom)
          (cons (cons num denom)
                (list-unusual-canceling-aux num (+ denom 1))))
         (else (list-unusual-canceling-aux num (+ denom 1)))))
  (list-unusual-canceling-aux 10 10))
```

(e) (The Knapsack problem.) You and a friend are going backpacking, and must bring along a bunch of equipment. You'd like to split the equipment between the two of you as evenly as possible. Write a SCHEME function to do this. Specifically, given a list of weights $(w_1\ w_2\ \ldots\ w_n)$, find a subcollection of the weights $(a_1\ a_2\ \ldots\ a_k)$ so that $\sum_i a_i$ is as close as possible to half the total weight of the w_i.

Solution:

```
(define (knapsack provisions)
  (define (total-weight provisions)
    (if (null? provisions)
        0
        (+ (car provisions)
           (total-weight (cdr provisions)))))
  (define target (/ (total-weight provisions) 2))
  (define (configurations provisions pack)
    (if (null? provisions)
        (list pack)
        (append
          (configurations
            (cdr provisions)
            (cons (car provisions) pack))
          (configurations (cdr provisions) pack))))

  (define (closest configs pack weight)
      (if (null? configs)
          pack
          (let ((next-pack-weight (total-weight
                                    (car configs))))
            (if (< (absolute (- next-pack-weight target))
                   (absolute (- weight target)))
                (closest (cdr configs)
                         (car configs) next-pack-weight)
                (closest (cdr configs) pack weight)))))
  (let ((configs (configurations provisions '())))
    (closest (cdr configs) (car configs)
             (total-weight (car configs)))))
```

(f) Define a SCHEME function `list-insert`, taking three arguments: a list, the new item, and the numerical position where you want the new item to be, starting at 1. So `(list-insert '(a b c) 'd 4)` evaluates to `(a b c d)`, and `(list-insert '(a b c) 'd 1)` evaluates to `(d a b c)`. This function should use no destructive modification.

Solution:

```
(define (list-insert lst item pos)
  (if (= pos 1)
      (cons item lst)
      (cons (car lst) (list-insert (cdr lst) item (- pos 1)))))
```

(g) Define a SCHEME function `list-insert!`, taking three arguments: a non-empty list, the new item, and the numerical position where you want the new item to be, starting at 1 (if the position is larger than the length of the list the item should be put at the end of the list. This should add the item using destructive modification, so any variable that refers to the list and any structure containing that list (the tail of an append, e.g.) will be changed. Example:

```
> (define a '(a b c))
> (define b '(d e f))
> b
(d e f)
> (define c (append a b))
> c
(a b c d e f)
> (insert-list! b 'z 1)
(z d e f)
> b
(z d e f)
> c
(a b c z d e f)
> (insert-list! b 'w 4)
(z d e w f)
> d
(a b c z d e w f)
```

Hint: draw a box-and-arrow diagram first.

Solution:

```
(define (list-insert! lst item pos)
  (cond ((= pos 1)
         (set-cdr! lst (cons (car lst)(cdr lst)))
         (set-car! lst item))
```

```
        ((null? (cdr lst))
         (set-cdr! lst (cons item '())))
        (else
         (set-cdr! lst (list-insert! (cdr lst)
                                     item
                                     (- pos 1)))))
  lst)
```

(h) (The Josephus Problem) The Josephus problem resembles the game of musical chairs, a version of the problem is as follows:

A travel agent selects n customers to compete in the finals of a contest for a two week vacation in Maui. The agent places the customers in a circle and then draws a number m out of a hat. the game is played by having the agent walk clockwise around a circle and stopping at every m^{th} customer and asking that customer to leave the circle, until only one remains. The one remaining customer is the winner.

Define a SCHEME function that takes a circular linked list as a parameter, generates a pseudo-random number and follows the Josephus algorithm to eliminate all but one node in the list and returns the value in the last remaining node in the list. Note: you will need to build a circular linked list by setting the cdr of a list to the list itself (using `set-cdr!` – you should probably write a function that takes a list and turns it into a circular list.

Solution:

```
(define (make-circular-list lst)
  (define (make-circular-list-aux lst head)
    (if (null? (cdr lst))
        (set-cdr! lst head)
        (make-circular-list-aux (cdr lst) head)))
  (make-circular-list-aux lst lst))

(define mylist (mlist 1 2 3 4 5 6 7 8 9 10 11 12))
(make-circular-list mylist)

(define (josephus lst n)
  (define (josephus-aux lst x prev)
    (cond ((= (car lst) (car (cdr lst))) (car lst))
          ((= x n) (begin (set-cdr! prev (cdr lst))
           ;;You can view the list after a removal
                          ;;(display (cdr lst))
                          ;;(display #\newline)
                          (josephus-aux prev 0 lst)))
          (else (josephus-aux (cdr lst) (+ x 1) lst))))
  (josephus-aux lst 0 lst))
```

3. Trees

```
(define (maketree v left-tree right-tree)
   (list v left-tree right-tree))

(define (value T) (car T))
(define (left T) (cadr T))
(define (right T) (caddr T))

(define (insert x T)
  (cond ((null? T) (make-tree x '() '()))
        ((eq? x (value T)) T)
        ((< x (value T)) (make-tree (value T)
                                    (insert x (left T))
                                    (right T)))
        ((> x (value T)) (make-tree (value T)
                                    (left T)
                                    (insert x (right T))))))
```

(a) Define a SCHEME function named `count-one-child` which returns the number of internal nodes of a binary search tree which have exactly one child.

Solution:

```
(define (count-one-child tree)
  (let ((left-child (left tree))
        (right-child (right tree)))
    (cond ((and (null? left-child)
                (null? right-child)) 0)
          ((and (not (null? left-child))
                (not (null? right-child)))
           (+ (count-one-child left-child)
              (count-one-child right-child)))
          ((null? left-child)
           (+ 1 (count-one-child right-child)))
          (else (+ 1 (count-one-child left-child))))))
```

(b) Define a SCHEME function named `count` which given a binary search tree and a value, returns the number of occurrences of the value in the tree.

Solution:

```
(define (occurances-in-tree T val)
   (if (null? T) 0
         (if (= (value T) val)
             (+ 1
```

```
                (occurances-in-tree (left T) val)
                (occurances-in-tree (right T) val))

            (+ (occurances-in-tree (left T) val)
                (occurances-in-tree (right T) val)))))
```

(c) Given a heap as defined in lecture, where all of the h-min values are positive, write a SCHEME function that returns the maximum value in the heap.

Solution:

```
(define (heap-max heap)
   (cond ((null? heap) -1)
         ((and (null? (left heap))(null? (right heap)))
          (h-min heap))
         (else (max (heap-max (left heap))
                    (heap-max (right heap))))))
```

4. Streams

(a) Write a function that, given a stream containing numbers $(a_1 a_2 \ldots)$, returns the stream $(b_1 b_2 \ldots)$ so that $b_i = a_{i+1} - a_i$. This should work for both finite and infinite streams, and should be written in terms of stream operators.

Solution:

```
(define (adj-diff str)
  (define (stream-subtract s1 s2)
    (cond ((null? s1)
           '())
          (else
           (cons-stream
             (- (stream-car s1)(stream-car s2))
             (stream-subtract (stream-cdr s1)
                              (stream-cdr s2))))))
  (cond ((null? str) '())
        ((null? (stream-cdr str)) '())
        (else (stream-subtract (stream-cdr str) str))))
```

(b) (Sieve of Eratosthenes) In ancient Greece, Eratosthenes gave the following algorithm for finding all prime numbers up to a specified number M:

1. Write down the numbers $2, 3, \ldots M$.
2. Cross out the numbers as follows:

(a) Keep 2 but cross out all multiples of 2 (i.e. cross out $4, 6, 8, \ldots$).

(b) Keep 3 but cross out all multiples of 3 (i.e. cross out $6, 9, 12, \ldots$).

(c) Since 4 is already crossed out, go on to the next number that is not crossed out.

(d) Keep 5 but cross out all multiples of 5.

Modifying this approach to streams, we can get the stream of all primes. Write a SCHEME function that generates the stream of prime numbers using the Sieve of Eratosthenes as follows: starting with the stream of all integers from 2 up, return `primes-from` that stream. `primes-from` takes a stream and cons-stream's the stream-car of that list with the result of calling `primes-from` on the stream with all multiples of the stream-car filtered out.

Note: we saw the code in lecture, but better if you write this from scratch and really understand how it works.

Solution:

```
(define (sieve p sequence)
  (cond ((null? sequence) '())
        ((= (modulo (stream-car sequence) p) 0)
         (sieve p (stream-cdr sequence)))
        (else
         (cons-stream (stream-car sequence)
                      (sieve p
                         (stream-cdr sequence))))))

(define (prime-stream)
  (define (primes-from str)
    (cons-stream (stream-car str)
                 (primes-from
                  (sieve (stream-car str)
                         (stream-cdr str)))))
  (primes-from (stream-cdr (stream-cdr ints))))
```

5. Objects

(a) Modify the following code from Lab 8 to require a password (by adding another parameter) for balance inquiries, deposits and withdrawals on the (new-account) object. The password should be set as an additional parameter to the new-account function.

```
(define (new-account initial-balance)
  (let ((balance initial-balance)
        (interestrate 0.01))
    (define (deposit f)
      (begin
        (set! balance
              (+ balance f))
        balance))
    (define (withdraw f)
```

```
    (begin
      (set! balance
            (- balance f))
      balance))
  (define (bal-inq) balance)
  (define (accrue) (begin (set! balance
                                (+ balance
                                   (* balance
                                      1
                                      interestrate)))
                          balance))
  (define (setrate r) (set! interestrate r))
  (lambda (method)
    (cond ((eq? method 'deposit) deposit)
          ((eq? method 'withdraw) withdraw)
          ((eq? method ' balance-inquire) bal-inq)
          ((eq? method 'accrue) accrue)
          ((eq? method 'setrate) setrate)))))
```

Solution:

```
(define (new-account initial-balance password)
  (let ((balance initial-balance)
        (interestrate 0.01)
        (pw password))
    (define (deposit f password)
      (cond ((equal? pw password)
      (begin
        (set! balance
              (+ balance f))
        balance))))
    (define (withdraw f password)
      (cond ((equal? pw password)
      (begin
        (set! balance
              (- balance f))
        balance))))
    (define (bal-inq)
      (cond ((equal? pw password) balance)))
    (define (accrue) (begin (set! balance
                                  (+ balance
                                     (* balance
                                        1
                                        interestrate)))
                            balance))
```

```
    (define (setrate r) (set! interestrate r))
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method ' balance-inquire) bal-inq)
            ((eq? method 'accrue) accrue)
            ((eq? method 'setrate) setrate)))))
```

(b) Add a method to the `new-account` object to change the password. It should take the old password and new password as parameters.

Solution:

```
(define (new-account initial-balance password)
  (let ((balance initial-balance)
        (interestrate 0.01)
        (pw password))
    (define (deposit f password)
      (cond ((equal? pw password)
      (begin
        (set! balance
              (+ balance f))
        balance))))
    (define (withdraw f password)
      (cond ((equal? pw password)
      (begin
        (set! balance
              (- balance f))
        balance))))
    (define (bal-inq)
      (cond ((equal? pw password) balance)))
    (define (accrue) (begin (set! balance
                                  (+ balance
                                     (* balance
                                        1
                                        interestrate)))
                            balance))
    (define (setrate r) (set! interestrate r))
    (define (change-password old new)
      (cond ((equal? old pw) (set! pw new))))
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method ' balance-inquire) bal-inq)
```

```
            ((eq? method 'accrue) accrue)
            ((eq? method 'setrate) setrate)
            ((eq? method 'change-pw) change-password)))))
```

(c) Write a SCHEME function that implements an object representing a rational number. It should use the GCD function (number b in the Tail Recursion question) to reduce the fraction represented by the object.

Solution:

```
(define (make-fraction numer denom)
  (define (GCD a b)
    (cond ((= a 0) b)
          ((= b 0) a)
          ((< a b) (GCD a (- b a)))
          (else (GCD (- a b) b))))
  (let ((numer (/ numer (GCD numer denom)))
        (denom (/ denom (GCD numer denom))))
    (define (value) (/ numer denom))
    (define (show) (begin (display numer)
                          (display " / ")
                          (display denom)
                          (display #\newline)))
    (lambda (method)
      (cond ((eq? method 'value) value)
            ((eq? method 'show) show)))))
```

(d) Define a SCHEME object to model a traffic-light that is always in one of four states, represented by the symbols 'red, 'yellow, 'green, and 'flashing-red. A traffic light can respond to four messages:

show: which returns the current state.

emergency! , which sets the state to 'flashing-red, regardless of what it was before.

set-light , which takes one argument – one of the symbols 'red, 'yellow, or 'green – and sets the state to that symbol.

cycle , which has no effect if the current state is 'flashing-red, but changes a 'green state to 'yellow, a 'yellow state to 'red, and a 'red state to 'green.

Solution:

```
(define (make-traffic-light)
  (let ((state 'red ))
    (define (emergency)
      (set! state 'flashing-red))
```

```
(define (cycle)
  (cond ((eq? state 'green) (set! state 'yellow))
        ((eq? state 'yellow) (set! state 'red))
        ((eq? state 'red) (set! state 'green))))
(define (set-light symbol) (set! state symbol))
(define (show) state)
(lambda (method)
  (cond ((eq? method 'cycle) cycle)
        ((eq? method 'set-light) set-light)
        ((eq? method 'emergency) emergency)
        ((eq? method 'show) show))))
```