Comp 210 – Intro to Programming and Computing EXAM #1 September 25, 2002
Rice University - Instructors: Wong & Barland NAME: ______________________

Instructions

1. This exam is conducted under the Rice Honor Code . It is a closed-notes, closed-book exam.
2. Fill in your name on every page of the exam.
3. You will not be penalized on trivial syntax errors, such as a missing parenthesis. Multiple errors or errors that lead to ambiguous code will have points deducted however.
4. In all of the questions, feel free to write additional helper methods or visitors to get the job done.
5. For each algorithm you are asked to write, 90% of the grade will be for correctness, and 10% will be for efficiency and code clarity.
6. There are a total of 100 points.
7. You have 50 minutes to complete the exam.
8. Use the back of the sheet the question is on if you need additional space.
9. You may write your functions using either forward or reverse accumulation unless otherwise directed.

Please write and sign the Rice Honor Pledge here:

1. (10 pts) **Design Templates**: What is the purpose of the design templates? How do they help us? In particular, relate them to the notions of variant and invariant entities.

2-3 sentences maximum!!

Answer:

The design templates are a part of a data-driven design methodology where the structure of the data naturally imposes an invariant structure on the functions that process them. The variant part of a function is the particular processing done to fulfill the particular needs of a given function.

2. (10 pts) **Critiquing and fixing existing code**: Consider the following data definitions:

```
;; A top represents a topping either
;; - a pep
;; - a veg.
```

```
;; pep represents a pepperoni topping and is a
;; structure with an area per slice in sq. inches and a calories per slice.
;; (make-pep num num)
(define-struct pep (area calories))

;; Example:
(define myPep (make-pep 2 50))
```

```
;; veg represents a vegetable topping is a structure with a color.
;; veg have no appreciable calories
;; (make-veg symbol)
(define-struct veg (color))

;; Example
(define myVeg (make-veg 'green))
```

```
;; a pizza is a structure with an area in square inches and a topping
;; (make-pizza num top)
(define-struct pizza (area top))

;; Examples
(define myP1 (make-pizza 200 myPep))
(define myP2 (make-pizza 200 myVeg))
```

Consider the following function:

```
;; nCals: pizza → num
;; Calculates the approximate number of calories in a pizza assumes the crust has 10 calories/sq. in.
;; and that the topping coverage is 80% of the total area.

(define (nCals p)
        ( + (* 10 (pizza-area p))
          ( cond
               [(pep? (pizza-top p))  (*
                       (/ (* .8 (pizza-area p)) (pep-area (pizza-top p)))  ;; This is the # of slices per pizza
                       (pep-cals (pizza-top p)))]
               [(veg? (pizza-top p)) 0])))

" nCals test cases:"
(= 6000 (nCals myP1))
(= 2000 (nCals myP2))
```

Even though the above function always yields the correct results, do you see anything wrong with its code body? If so, please explain clearly and completely **what** you feel is improperly implemented about this code and **why** it is improper. Assume that you are given the data definitions as well as the contract, purpose and header for the function and thus have no control over them.

2-3 sentences maximum!!

Answer:

It violates the encapsulation of the topping structure by 1) checking for the actual type of topping and 2) by then pulling out the internal data of a topping. The use of "magic numbers" is also bad programming practice because it lowers the abstraction level.

3. (35 pts total) **Lists and Forward and Reverse Accumulation**: Consider the following data definition:

```
;; A list-of-symbol, "los," is either
;; - empty
;; - (cons [symbol] [list-of-symbol])
```

a. (10 pts) Write the design template for a list-of-symbol ("los").

Answer:

```
(define (f-los a-los …)
    (cond
        [(empty? a-los) …]
        [(cons? a-los) …(first a-los)…(f-los (rest a-los) …)…]))
```

b. (10 pts) Write a function that counts the number of occurrences of a given symbol in a list-of-symbol ***using natural recursion (reverse accumulation).*** You are given the contract, test cases below and you may omit the purpose. Call your function "nSyms". Be sure to follow your own template!

;;nSyms: list-of-symbol, sym → num

Test cases:
```
(define myLOS (list 's 'g 's 'w 't 's 'g))
(= 0 (nSyms empty 's))
(= 3 (nSyms myLOS 's))
(= 2 (nSyms myLOS 'g))
(= 0 (nSyms myLOS 'z))
```

Answer:

```
(define (nSyms a-los s)
    (cond
      [(empty? a-los) 0]
      [(cons? a-los)  ( + (cond
                              [(symbol=? s (first a-los)) 1]
                              [else 0])
                          (nSyms (rest a-los) s))]))
```

Alternatively:

```
(define (nSyms a-los s)
    (cond
      [(empty? a-los) 0]
      [(cons? a-los)  (cond
                          [(symbol=? s (first a-los)) (+ 1 (nSyms (rest a-los) s))]
                          [else 0])  (nSyms (rest a-los) s))]))
```

(15 pts) Implement the above function ***using an accumulator style algorithm (forward accumulation).*** You may omit the contract, purpose and test cases. Call your function "nSyms2".

Answer: Any of the following 3 main functions plus the helper:

```
(define (nSyms2 a-los s)
    (cond
        [(empty? a-los) 0]
        [(cons? a-los)  (nSyms2_help (rest a-los) s
                                     (cond
                                        [(symbol=? s (first a-los)) 1]
                                        [else 0]))]))

(define (nSyms3 a-los s)
    (cond
        [(empty? a-los) 0]
        [(cons? a-los)  (nSyms2_help_help a-los s 0)]))

(define (nSyms4 a-los s)
     (count_nSyms2 a-los s 0))

;; nSyms2_help: los, sym, num → num
;; returns the number of occurrences of s in the los added to the given accumulator.
(define (nSyms2_help a-los s acc)
    (cond    [(empty? a-los) acc]
             [(cons? a-los)  (nSyms2_help (rest a-los) s
                                          (+ acc (cond
                                                    [(symbol=? s (first a-los)) 1]
                                                    [else 0])))]))
```

4. (30 pts total) **List of Lists**: Consider a list that contains lists. For example, suppose we had the weights of each person in a number of different classes. We'll call this a list of class weights or "LoCW" It could be represented as such:

```
(define myLoCW (list (list 98 120 104 230) (list 280 134 260 155 101) empty (list 132 201 143)))
```

Note that empty and (cons empty empty) are both examples of a LoCW as well.

Warning: THINK SIMPLY!! Hint: See Problem #2.

a. (10 pts) Write the data definition(s) and template(s) for a list of class weights (LoCW) where class weights (CW) is a list of weights of the students in a class. Be sure to also write the data definitions for any structures that LoCW may contain. Assume the function is recursive and include anything additional that can be said about recursive functions on these data types.

Answer:

```
;; A LoCW is either
;; - empty
;; - (cons [CW] [LoCW])

(define (f-LoCW a-LoCW …)
      (cond [(empty? a-LoCW) …]
             [(cons? a-LoCW) …(f- CW (first a-LoCW)…)…(f- LoCW (rest a-LoCW)…)…]))

;; A CW (class weights) is either
;; - empty
;; - (cons [num] [CW]), where num is a weights in lbs.

(define (f-CW a-CW …)
      (cond [(empty? a-CW) …]
             [(cons? a-CW) …(first a-CW)…(f- CW (rest a-CW)…)…]))
```

b. (20 pts) Write a function that will return the total weight of all the people in all the classes. Be sure to include the contract, purpose and header for any additional functions you write. *Follow your template(s)!*

```
;; total_weight: LoCW → num
;; Calculates the total weight of  a LoCW.
(define (total_weight a- LoCW) …    ;; finish this below

"total_weight test cases:"
(= 1958 (total_weight myLoLoW))
(= 0 (total empty))
(= 0 (total (cons empty empty)))
```

Answer:

```
(define (total_weight a-LoCW)
      (cond [(empty? a-LoCW) 0]
               [(cons? a-LoCW) (+ (class_weight (first a-LoCW))
                                              (total_weight (rest a-LoCW)))]))

;; class_weight:  CW → num
;; Calculates the total weight of a class weights.
(define (class_weight a-CW)
      (cond
            [(empty? a-CW) 0]
            [(cons? a-CW) (+ (first a-CW) (class_weight (rest a-CW)))]))
```

5. (15 pts) **Natural Numbers**:

Given a natural number, n, write a function, nPerfects, that will return how many of numbers in the set {n, n-1, n-2, …, 1} are "perfect". That is, it will count the perfect numbers in that set.

It is not important what it means for a number to be "perfect" because a "friend" (yeah, right) *has already written* a function perfect? for you. That is, you are given

```
;; perfect?: natNum → boolean
;; returns true if the given number is perfect
(define (perfect? n)….)        ;; You are already given this code, so don't write this function!

"perfect? test cases:"
(boolean=? false (perfect? 0))
(boolean=? true (perfect? 6))
(boolean=? true (perfect? 28))
(boolean=? false (perfect? 250))
(boolean=? true (perfect? 496))
(boolean=? false (perfect? x)) ;; for any x < 10000 except 6, 28, 496 and 8128
```

Write the function nPerfects including its contract, header, purpose and test cases. (Though not required, writing the template for natural numbers may helpful.)

Answer (accumulator style implementation also acceptable):

```
;; nPerfects: natNum → num
;; counts all the perfect numbers less than or equal to n
(define (nPerfects n)
  (cond
    [(zero? n) 0]
    [(> n 0) (+ (cond        ;; "else" acceptable
                  [(perfect? n) 1]
                  [else 0])
              (nPerfects (sub1 n)))]))

"nPerfects tests:"
(= 0 (nPerfects 0))     ;; must have
(= 1 (nPerfects 6))      ;; must have
(= 2 (nPerfects 28))      ;; must have
(= 2 (nPerfects 250))      ;; for example
(= 3 (nPerfects 496))      ;; must have
(= 4 (nPerfects 10000))     ;; for example

;; Accumulator version (one possibility):
(define (nPerfects2 n)
  (cond
    [(zero? n) 0]
    [(> n 0)  (nPerfects2_help (sub1 n) (cond
                                 [(perfect? n) 1]
                                 [else 0]) ) ]))

;; nPerfects2_help:  natNum num --> num
;; adds the acc to the count of perfect numbers les than or equal to n
(define (nPerfects2_help n acc)
  (cond
    [(zero? n) acc]
    [(> n 0)  (nPerfects2_help (sub1 n) (+ acc (cond
                                               [(perfect? n) 1]
                                               [else 0]) ))]))
```