Comp 210
Exam #1
1998 Fall

Name: ______________________________

This is a closed-book, closed-computer, three hour exam, due 98.Oct.14 (Wed), 10:00. This should be plenty enough time. **Make sure to read each problem carefully.** If you need to make any reasonable assumption to do a problem, just state that in the margin. Please refrain from discussing the exam until after the due-date.

Except as indicated, you may (only) use the following built-in Scheme functions: *symbol? eq? boolean? empty? cons? cons first rest list number? zero? positive? negative? add1 sub1* + - * / = < > $\leq$ $\geq$ and the special forms *lambda define cond define-struct and or*.

Include contracts for all functions you write.

Time started:____________

Time finished:____________

Pledge:

Problem 1	15	
Problem 2	20	
Problem 3	25	
Problem 4	20	
Problem 5	20	
Total	100	

1. (15 pts) In the kingdom of Tuhundredtenia, citizens whose income is less than \$210 are taxed 2% of their income. Those is income is more are taxed 2% of that first \$210, and 10% of the rest of their income. Write a function `tax: number` $\rightarrow$ `number` which given an income, computes the tax due. Naturally, changing any of the above the numbers should require changing only one number in your solution.

2. (20 pts) List Templates

   (a) Write the data definition, three examples, and the template for *List-of-Number*.

   (b) Consider the function *insert: number, List-of-Number* $\rightarrow$ *List-of-Number* which takes in a number and an *ordered* (non-decreasing) list of numbers, and returns a list with the number inserted into the right place in the list. For example, `(insert 8 (list 2 4 6 10)) = (list 2 4 6 8 10)` (A.) Give three other examples, and (B.) write the function.

(c) Consider the function *insert-sort: List-of-Number* $\rightarrow$ *List-of-Number*, which takes its unordered input, and returns the same elements in a sorted (non-decreasing) list. For example, `(insert-sort (list 8 6 2 4 10)) = (list 2 4 6 8 10)` (A.) Give two other examples, and (B.) write the function. (Be sure to follow the template!)

3. (25 pts) An *Employee* is either a secretary (who has a name and salary), or a manager (who has a salary, and a list of employees who they directly supervise).

   (a) Write all a data definition for *List-of-Employee*, and for *Employee*. If these data definitions use any `make-`*someStruct*s, include appropriate `define-struct`s.

   (b) Give four different examples of *Employee*s.

(c) Write a template for *Employee*, and one for *List-of-Employee*.

(d) Consider the function *cost-to-company: employee* $\rightarrow$ *number*, which calculates the salary of that employee and any employees *anywhere* under their supervision (even indirectly). Next to your sample employees above, (A.) write in what *cost-to-company* returns on each of them. (B.) Write the function.

4. (20 pts) Recall that a *NatNum* is either

- `zero`, or
- `(add1` *NatNum*`)`.

(a) Write the template for a function which handles two *NatNums*.[1]

(b) An egalitarian computer virus, wanting to get rid of all inequality,[2] has destroyed all computer implementations of the functions $<, \leq, \geq, >, -, abs$. In fact, the only surviving numerical functions are $add1, sub1, zero?, positive?$. Save the day by writing *greater?: NatNum,NatNum* $\rightarrow$ *boolean*, which returns whether or not its first argument is bigger than its second.

[1] If your template is written such that the *order* in which you list the cases is essential to *covering* all cases, make a comment to this effect.

[2] Presumably somebody who felt that Tuhundredtenia's tax code didn't fairly account for cost-of-living.

5. (20 pts) The following code is the straightforward way of calculating the sum of a list of numbers with and without an accumulator.

```
;; sum: list-of-number --> number
;;
(define sum
  (lambda (data)
    (cond [(empty? data) 0]
          [(cons?  data) (+ (first data) (sum (rest data)))])))


;; sum-accum-helper: list-of-number, number --> number
;; Return the sum of sum-so-far and the numbers in data
;;
(define sum-accum-helper
  (lambda (sum-so-far data)
    (cond [(empty? data) sum-so-far]
          [(cons?  data) (sum-accum-helper (+ sum-so-far (first data))
                                           (rest data))])))

;; sum-accum: list-of-number --> number
;; Return the sum of the numbers in nums-to-add.
;;
(define sum-accum
  (lambda (nums-to-add)
    (sum-accum-helper 0 nums-to-add)))
```

Hand evaluate the following expressions, showing the first two steps of the evaluation in complete detail, and then showing (only) the intermediate expressions just before calling one of the above-defined functions. Be careful not to drop/add parentheses. (Of course, if you want to claim that two expressions have equal values, use an equal-sign between them.) To save a bit of writing, you may write `sum-ah` as shorthand for `sum-accum-helper`.

- `(sum (list 7 2.718 3.14))`

- `(sum-accum (list 7 2.718 3.14))`

Yes or no: Do *sum* and *sum-accum* always return the same answer, for all possible inputs?