Comp 210
Exam #2
1998 Fall

Name: ________________________________

This is a closed-book, closed-computer, *3.5 hour* exam, due 98.Dec.02 (Wed), 10:00.

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* + - * / = < > ≤ ≥ and the special forms *lambda define cond define-struct and or* (plus functions created by *define-struct*). Furthermore, you also use the special forms *local, set!, begin.*

Include contracts for all functions you write, if not provided in the problem.

Time started:____________

Time finished:____________

Pledge:

Problem 1	15	
Problem 2	20	
Problem 3	20	
Problem 4	15	
Problem 5	20	
Problem 6	10	
Total	100	

1. (15 pts) Given a number n, how many terms of the doubling sequence $1, 2, 4, 8, 16, \ldots$ must be looked at until one is $\geq n$? One way to calculate this is to work backwards: count how many comparisons we must make until n successively halved becomes ≤ 1:

```
;; times-halved: number --> natNum
;;
(define (times-halved n)
  (if (<= n 1)
      1
      (add1 (times-halved (/ n 2)))))

; Test:
(times-halved 2) = 2
(times-halved 6) = 4
(times-halved 8) = 4
(times-halved 1/7) = 1
```

An equivalent, but slightly different, approach would be to work forwards: count how many comparisons we must make until 1 successively doubled becomes $\geq n$. Implement this approach by writing a function `times-doubled` using an accumulator, so that it is tail-recursive. (Do not use `set!`.)

2. (20 pts) We will use the following structure in this problem and the next.

```
(define-struct student (name id friend))
; where <name> is a symbol, <id> is a natNum, and <friend> is a student or #f.
```

Write a function `create-student: symbol --> student` which takes a name, and creates the corresponding student structure. Initially the `friend` field is `#f`. The interesting part is the `id`: The first student created should have an `id` of 1, whe next an `id` of 2, the nexts with `id`s of 3,4,.... (So your function must remember how many times it has previously been called.) Note that while somebody could later use `set-student-id!` to fake a student's id, there should be no way they could cause `create-student` to give an incorrect id[1].

Hint: Define the function inside a `local` (perhaps giving it a silly name); have the `local` return that function, giving it a proper name outside the define.)

[1]To think about only: using what we know, it is possible to make the function `set-student-id!` inaccessible to people calling your function. How? (3pts extra credit if you answer this *after* completing all the other exam questions.)

Write the function `befriend: student, student --> (void)`[2], which sets the first student's friend to be the second student. (Note that alas, the second student's friend does not automatically become the first student.)

What functions would you call, to represent students Chris (whose friend is Pat), Pat (whose friend is Kim), and Kim (whose friend is Pat). Draw a picture which also includes Jo (whose friend is Kim) (show no code for for Jo).

[2] You may return the symbol `'pal` instead of `(void)`, if you prefer.

3. (20 pts)

   At Nepotism State University, the child of the most generous alumnus is guaranteed to be assigned to live in a suite with their friend, their friend's friend, their friend's friend's friend, etc. (In the previous example, if Chris' parent donated enough money, Chris would be in a suite with both Pat and Kim, but not Jo.)

   Write a function which accepts a student, and returns the smallest size suite in which they can be housed, if their parents pay up. In order to avoid loops, go back and add a field `assigned?` to the student structure, and use this field to ensure finishing.

   You may assume that every student has a friend (that is, the friend field is never `#f` when this function is called).

   What would you need to do, so that your function could be called on several potential donor's children? (State exactly what function(s) you would add/modify, to effect this.) Describe your approach *concisely* (code not required).

4. (15pts) Many programming languages include other looping constructs in addition to a for-loop. One such construct is "do-until", which works as follows:

   - Syntax: do-until involves a body and a condition, e.g. `(do-until` *body!* *condition*`)`. In scheme, we'll make *body!* and *condition* both be functions that take zero arguments.
   - Semantics (meaning): Execute (evaluate) *body!*, and then test the *condition*: if it returns true, then the loop terminates, otherwise the whole process is repeated.
   - Pragmatics: Note that unlike a for-loop, the body is always executed at least once. This tends to be handy in loops involving input/output, where you want to at least read one input before quitting. **Example**: the main loop of the maze-program could be written:

     ```
     (do-until (lambda () (handle-command! (read)))
               (lambda () done-yet?))
     ```

     (Presumably, the variable `done-yet?` is initially false, and `handle-command!` might `(set! done-yet? #t)`.)

   Write `do-until` in Scheme. (Remember to include the contract.)

   Using `do-until`, rewrite `timesDoubled`, from Problem 1 (now you may use `set!`).

5. (20 pts) In Jam2000,

- What assembly instruction is encoded as 00204335?
- What is a machine encoding of the assembly instruction (st R4 R3)?
- Translate the following (tail-recursive) Scheme function to JAM assembly language. Indicate where in memory each instruction is stored, and include comments about what each (group of) lines does. (This program is able to fit all its variables into registers; it does not need to load/store from main memory.) Instead of returning a value, your assembly program should print the answer (in this case, it will be 32).

```
;; Return result * 2^k.
;;
(define (powerOfTwo k result)
  (if (zero? k)
      result
      (powerOfTwo (sub1 k) (* 2 result))))

(powerOfTwo 5 1)  = 32 ; Two to the fifth; 1 is initial accum value.
```

6. (10pts) Write `map`, using `make-list-fun`.

   As reminders:

   - `map` takes a function f and a list *lyst*, and returns a list containing the f applied to each element of *lyst*. For example, `(map sqrt (list 16 1 25)) = (list 4 1 5)`.
   - ```
     (define make-list-fun
       (lambda (base combine)
         (local [(define the-fun
                   (lambda (lyst)
                     (cond [(empty? lyst) base]
                           [(cons?  lyst) (combine (first lyst)
                                                   (the-fun (rest lyst)))])))]
           the-fun)))
     ```
     ```