COMP 210: Principles of Computing and Programming
Second Exam (Solution Key)

November 13th, 2006 (Fall 2006)

Name: ______________________ (please write at top of each page before you start)
User id: ______________________
Honor code pledge:

____

____

____

This exam is closed book, closed notes, closed computer.

Score sheet (to be completed by grader)

Problem	**Max**	**Score**
1	**25**	
2	**25**	
3	**25**	
4	**25**	
Total	100	

Problem 1. (25 points) Consider the following definition for a binary tree:

```
; A binary search tree of numbers (bst) is one of
; - false, or
; - (make-node n l r),
;   where n is a number l and r are each bsts.
; All numbers in l are less than or equal to n, and
; all numbers in r are greater than n.

(define-struct node (n l r))
```

You have decided that you need a function `insert : bst number -> bst` that takes a binary search tree and a number and returns a new binary search tree where the new number has been inserted in an appropriate place. Which template would you use?

1. (10 pts) Write down the template for a function processing a `bst`.

   Solution:

   ```
   (define (f bst)
     (cond
       [(boolean? bst)...]
       [(node? bst) ...(node-n bst)...(f (node-l bst))...(f (node-r bst))...]))
   ```

   Common Problems:

   - Testing empty? or symbol? instead of boolean?: -3
   - Using false? instead of boolean?: -1
   - Including extra parameters or cases: -2 per
   - Failing to include recursive definition: -2
   - Not including node-n in the template: -2

2. (15 pts) What is the code for the `insert` function?

   Solution:

```
(define (insert bst n)  (cond
    [(boolean? bst) (make-node n false false)]
    [(node? bst)
     (cond
       [(<= n (node-n bst))
        (make-node
          (node-n bst)
          (insert (node-l bst) n)
          (node-r bst))]
       [else
        (make-node
          (node-n bst)
          (node-l bst)
          (insert (node-r bst) n))])]))
```

   Common Problems:

   - Using lists instead of the defined tree structure: -5
   - Not using node-l, node-n, node-r: -3
   - No answer: -15
   - Using empty instead of false (-0 if used empty? in template)
   - Returning false instead of (make-node nat false false): -5
   - Returning nat instead of (make-node ...): -3
   - Returning the result of insert directly without saving the other half of the tree: (e.g. `(insert (node-l abstr) num)` instead of `(make-node (insert node-l abstr) num)` -7 II
   - Attempting to insert the num at the top of the tree: -1 (when with above) I
   - Not checking for node-n=num: -3

Problem 2. (25 points) You are the Chief Technical Officer (CTO) of Correctness Is Us (CIU), an innovative software development firm that specializes in producing correct software solutions quickly. CIU was contracted by an up-and-coming financial firm to build a stock-trading agent. Looking into the details of the job, you realize that you need a function `lookup` that takes `predictions` as a list of numbers, `prices` as a list of numbers, and today's `prediction` as a number. If `prediction` is the ith element in `predictions`, `lookup` returns the ith element in `prices`. If `prediction` is not in `predictions`, the function returns `false`.

Both lists are sorted in increasing order, and are of equal length; `predictions` contains no duplicate values.

1. (5 pts) What is the basis for induction in this problem?
2. (20 pts) Write the code for `lookup`. Do **not** use auxiliary functions or built-in Scheme functions. Once you have written your answer, review it to make sure that it does not perform more work than necessary.

Solution: For the first part, the induction is lock-step on the size of the lists `predictions` and `prices`. Our base cases are either finding the prediction or the empty list. [Only the first part is required]

For the second part,

```
(define (lookup predictions prices prediction)
  (cond
    [(empty? predictions) false]
    [(cons? predictions)
      (cond
        [(= (first predictions) prediction) (first prices)]
        [else (lookup (rest predictions) (rest prices) prediction)])]))
```

Common Problems:

- -5: No mention of induction or recursion (12 instances)
- -5: Left blank (5 instances)
- -3: Not specifying what is recurred on
- -3: Confusing > with < (4 instances)
- -7: Does not short-circuit based on sorted order (weighted heavily because it's one of the key points of the problem, and is hinted at directly (20 instances)
- -7: Implemented using auxiliary functions and numeric indices (weighted heavily because the instructions specifically prohibit aux functions)

Problem 3. (25 points) Long before you became the CTO of CIU, you worked there as a summer intern developing a trading agent for Radex Investments. During that project, you found out that a lot of data is communicated in the Extensible Markup Language (XML), and that XML is a lot like nested lists. But first the information comes in a very flat form. For example, a ticker stream (`ts`) is basically a list of symbols or numbers (that is, `[symbol or number]`). But it has an important additional constraint: each news item in the stream, while consisting of multiple symbols or numbers, follows a specific format. There are two kinds of news items:

- A forecast, which begins with the symbol `'Forecast`, and is followed by a source (a symbol), a stock name (a symbol), and a prediction (a number).
- An actual price report, which starts with the symbol `'Price`, and is followed by a stock name (a symbol) and a price (a number).

To make this data more manageable, you decide to parse the ticker stream into a more usable form (that is much more like XML). In particular, you design the type

```
; A news item (ni) is either
;  (list 'Forecast source stock prediction)
;  (list 'Price    stock price)
; where source and stock are symbols, and
;       prediction and price are numbers
```

and you decide to write a function `parse-news: [symbol or number] -> [ni]`.

1. (5 pts) What template would you use for `parse`? Describe the type of the template, and write out its code.

   Solution: We use the generative recursion template.

   ```
   (define (generative-recursive-fun problem)
     (cond
       [(trivially-solvable? problem)
        (determine-solution problem)]
       [else
        (combine-solutions
          ... problem ...
          (generative-recursive-fun (generate-problem-1 problem))
          ...
          (generative-recursive-fun (generate-problem-n problem)))]))
   ```

   Common Problems:

   - -5: No mention of generative recursion (33 instances) (No one correctly wrote out the generative recursion template. One correctly stated that it used gen. recursion, and one hinted at the answer by using the word "combine" in his template.)

2. (10 pts) What auxiliary functions will you use to decompose the input? Provide both the types (contracts), purpose statement (brief), and code.

   Solution:

```
; news-item-first: ts -> ni
; Parse the first news item in a ts.  Assumes the stream is
; structured correctly and nonempty.
(define (news-item-first ts)
  (cond
    [(symbol=? 'Forecast (first ts))
     (list (first ts) (first (rest ts)) (first (rest (rest ts)))
           (first (rest (rest (rest ts)))))]
    [(symbol=? 'Price (first ts))
     (list (first ts) (first (rest ts)) (first (rest (rest ts))))]))

; news-item-rest: ts -> ts
; Get the "rest" of a token stream after its first news item.  Assumes
; the stream is structured correctly and nonempty.
(define (news-item-rest ts)
  (cond
    [(symbol=? 'Forecast (first ts)) (rest (rest (rest (rest ts))))]
    [(symbol=? 'Price (first ts)) (rest (rest (rest ts)))]))
```

   Common Problems:

   - -3: No “first” equivalent, returning the first ni
   - -3: No “rest” equivalent, returning the rest of the stream after the first ni
   - -2: Not using the ni constructors that follow from the definition (that is, e.g., (list 'Forecast source stock prediction))
   - -1 to -3: Signature does not match implementation
   - -1 to -4: Broken code
   - -1: Unnecessary reliance on mutual recursion with the parse function
   - -1: Handling cases that will never occur

3. (10 pts) What is the code for your parse function?

   Solution:

```
(define (parse ts)
  (cond
    [(empty? ts) empty]
    [(cons? ts) (cons (news-item-first ts) (parse (news-item-rest ts)))]))
```

   Common Problems:

   - -4: Recurred on "(rest ts)" rather than skipping to the next news item in the stream
   - -1 to -4: Broken code
   - -1: Handling cases that will never occur
   - -1: Confusing "append" and "list" with "cons"

Problem 4. (25 points) The following code was used to find a path in a graph:

```
(define (find-route g start stop)
  (cond [(= start stop) (cons start empty)]
        [else (local ((define possible-route
                        (find-route/list g
                                         (neighbors g start)
                                         stop)))
                (cond [(boolean? possible-route)
                       false]
                          [else
                       (cons start possible-route)]))]))

(define (find-route/list g start-list stop)
  (cond [(empty? start-list)
              false]
           [else
              (local ((define possible-route
                           (find-route g (first start-list) stop)))
              (cond [(boolean? possible-route)
                           (find-route/list g (rest start-list) stop)]
                        [else
                           possible-route]))]))
```

A simple way to let this code handle cyclic graphs it limit the number of hops an acceptable path can have.

1. (20 pts) Modify the functions to take an extra argument `budget` and use it to limit the length of any path explored by the search.
2. (5 pts) When you call the search for the first time, what value should you use for budget?

Solution:

```
(define (find-route g start stop budget)
  (cond [(= budget 0) false]
        [(= start stop) (cons start empty)]
        [else (local ((define possible-route
                        (find-route/list g
                                         (neighbors g start)
                                         stop
                                         budget)))
                (cond [(boolean? possible-route) false]
                      [else (cons start possible-route)]))]))
```

```
(define (find-route/list g start-list stop budget)
  (cond [(empty? start-list) false]
            [else
               (local ((define possible-route
                              (find-route g
                                                (first start-list)
                                                stop
                                                (sub1 budget))))
               (cond [(boolean? possible-route)
                              (find-route/list g
                                                  (rest start-list)
                                                  stop
                                                  budget)]
                        [else possible-route]))]))
```

The initial budget should be equal to the number of vertexes in the graph. There is no need to go through the same vertex twice in a path: we can simply take out the loop and get a shorter path. Technically the number of vertexes - 1 is sufficient.

Common Problems:

- Not checking budget in proper location: -8
- Checking something other than budget is zero (or budget is size of graph): -6
- Checking the budget too late: -8
- Returning an error instead of false: -4
- Not terminating: -8
- Subtracting too often: -4
- Failing to pass budget to find-route or find-route/list: -1
- Failing to add budget to the declaration of find-route or find-route/list: -1
- Passing in other parameters: -1
- No answer or 'whatever you want': -5