COMP 210: Principles of Computing and Programming
First Exam

Solution Key

September 26th, 2007 (Fall 2007)

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	**10**	
2	**10**	
3	**30**	
4	**20**	
5	**30**	
Total	100	

Problem 1. (10 points) Given the following definition for the function `f`

```
(define (f a)
        (cond [(and (symbol? a)  (= 'a 1)) (or false (/ 1 0))]
              [(or  (number? 'a) (= a 2))  (/ a 0)]))
```

hand evaluate the expression `(f 2)`. Make sure that you perform evaluation in the same way that DrScheme evaluates programs.

Solution:

```
  (define (f a)
          (cond [(and (symbol? a)  (= 'a 1)) (or false (/ 1 0))]
                [(or  (number? 'a) (= a 2))  (/ a 0)]))
  (f 2)

= (cond [(and (symbol? 2)  (= 'a 1)) (or false (/ 1 0))]
        [(or  (number? 'a) (= 2 2))  (/ 2 0)]))     //Substitution: 2 points

= (cond [(and false (= 'a 1)) (or false (/ 1 0))]   //symbol?: 1 point
        [(or  (number? 'a) (= 2 2))  (/ 2 0)]))

= (cond [false (or false (/ 1 0))]                  //(and false...): 1 points
        [(or  (number? 'a) (= 2 2))  (/ 2 0)]))

= (cond [(or  (number? 'a) (= 2 2))  (/ 2 0)]))     //cond [false ...: 1 points

= (cond [(or false (= 2 2))   (/ 2 0)]))            //number?: 1 points

= (cond [(or false true)   (/ 2 0)]))               //(= 2 2): 1 points

= (cond [true (/ 2 0)])                             //(or #f #t...): 1 points

= (/ 2 0)                                           //cond [true ...: 1 points

= / : divide by zero                                //Correct answer: 1 points
```

Common Problems:

- Skipping steps. If a step was not explicitly performed, but it was obvious that the step was performed correctly due to later steps, then only some of the points were lost. If there was no way to determine if the step was performed correctly, then all of the points were deducted.
- Substituting 2 for the symbol 'a, or failing to substitute 2 for a until later in the evaluation.

- Continuing to evaluate (and false (= ‘a 1)) to (and false false).
- Failing to throw away the false part of a cond.

Problem 2. (10 points) A value is a Scheme program that cannot be reduced any further. Which of the following are *values*? Write YES or NO next to each one.

Solution: Interleaved with rest of problem statement. Each sub-problem was worth 1 points.

- `'x`
  - Yes. Any symbol is a value.
- `(cons (f 5) empty)`
  - No. This term is not a value because `(f 5)` still needs to be reduced.
- `(cons 5 6 empty)`
  - No. Cons takes two arguments, so, this is not even a syntactically valid expression.
- `false`
  - Yes. This is one of the two values used for booleans.
- `(rest (cons 5 empty))`
  - No. Rest computes the tail of a list. Here it returns `empty`.
- `(make-posn 5 6)`
  - Yes.
- `(and true true)`
  - No. And is a primitive operation. This term evaluates to `true`.
- `(cons (cons empty empty) empty)`
  - Yes. This is an example value for a list of lists.
- `(cons false empty)`
  - Yes. This is a list of booleans, and the list has only one element.
- `(cons empty false)`
  - No. Remember that cons actually takes its value to make sure that its second argument is a list.

Problem 3. (30 points) An *odd list (OL, for short)* is a peculiar data type. For example, an OL can never be empty, and it can never contain an even number of elements. This way, any OL will have an odd length. A customer asked that we define and use our own OL data type in the software we build for their project. The OL data type must be either one element, or a structure consisting of two elements and another odd list.

i. Write down a data type definition for an odd list of numbers (OLN). Define and use a different structure for each variety in your data type definition.

   – 10 points

```
(define-struct OLE (head)                ; an odd-list-element
(define-struct OL (head-1 head-2 tail)) ; an odd-list

; an OLN is either
;   (make-OLE n)
;   (make-OL n1 n2 oln)
; where n, n1, n2 are numbers and oln is a OLN
```

ii. Write down three different examples of an OLN

   – 10 points

```
(define oln1 (make-OLE 0))
(define oln2 1 2 oln1)
(define oln3 1 2 oln2)
```

iii. Write down the template for a function that uses an OLN

   – 5 points

```
; f : OLN -> ...
; (define (f x)
;   (cond [(OLE? x) ... (OLE-head x) ... ]
;         [else     ... (OL-head-1 x) ... (OL-head-2 x) ...
                         (f (OL-tail x)) ...]))
```

iv. Write down a function that counts the non-zero elements in a given OLN

   – 5 points

```
; count : OLN -> number
(define (count x)
  (cond [(OLE? x) (count-non-zero (OLE-head x))]
        [else     (+ (count-non-zero (OL-e1 x))
                     (count-non-zero (OL-e2 x))
                     (count (OL-ol x)))]))
```

```
; count-non-zero : number -> number
(define (count-non-zero x)
  (if (> x 0) 1 0))
```

Common Problems:

- In (i), Not defining OLN using structures
- In (ii), Inconsistency between examples and data type definition
- In (iii), Inconsistency between template and data type definition
- In (iii), When handling the recursive variety of OLN, not referring to both the first and second elements
- In (iv), Recurring on non-OLNs (a *strong* hint of an inferior solution)
- In (iv), Not abstracting multiple tests for non-zero

Problem 3. (Additional space for your solution)

Problem 4. (20 points)

i. Is it possible to define the `list` construct as a function? If yes, give the definition. If not, explain why.

- 5 points

  We do not yet know how to define `list` as a function because it takes an arbitrary number of arguments.

ii. Expand `'((1 2) (list 3 4) 5)` into an expression that uses only `empty`, `cons`, numbers, booleans and symbols (as needed).

- 15 points

  As an intermediate step we can expand the above to:

```
(list (list 1 2)
      (list 'list 3 4)
      5)
```

  From this we can get to the final answer:

```
(cons (cons 1 (cons 2 empty))                  // Line 1
  (cons (cons 'list (cons 3 (cons 4 empty)))   // Line 2
    (cons 5 empty)))                           // Line 3
                                               // Line 4
                                               //  parens
                                               //  'list
```

Problem 5. (30 points)

Respond to the following questions with a brief (2-4 sentence) answer.

i. The design recipe that we have learned about in this course has several components. One of these components is the design template. What is the purpose of the design template? How does it help us solve problems?

- 6 points

  The design template makes explicit the strong connection between data structure definitions and program design. In other words, data design drives program design.

ii. As part of our problem-solving skills, we learned guidelines for using variable and function declarations. When is it a good idea to define a separate function?

- 6 points

  We have seen two fundamental reasons for using separate functions. First, we should use a separate function for each distinct data type that we use; second, we should abstract common expressions into functions.

iii. How do Scheme functions differ from mathematical functions?

- 6 points

  One thing to note about Scheme functions is that order can be significant. For example, when applying a function, argument are evaluated in left-to-right order. Also the Scheme functions `and` and `or` use short-circuited evaluation. (We will learn more about modelling this later.)

  More deeply, Scheme functions can cause side effects such as generating exceptions and (as we will see later) changing the values of variables. When combined with order, programs with side-effects can have very subtle behaviors!

  It should be noted that while Scheme's syntax is 'prefix' and while we do commonly write mathematical algebraic functions in 'infix' notation, this is really an artificial distinction. It should also be noted that mathematical functions can operate over non-numerical data, two examples being sets or logical values.

iv. What two different ways did we study for defining natural numbers? What are the advantages of each of the two representations?

- 6 points

```
; a natural is
;   0
;   (add1 n), where n is a natural
;
; a Natural is
;   'Zero
;   (make-nat n), where n is a Natural
```

The first definition represents naturals using Scheme's internal representation and consequently exploits Scheme's built-in natural number operators. In contrast, the second definition represents a natural number by the length of a list (or a struct-list) and in fact allows us to define the naturals from 'scratch'. For this second kind of natural, we must define our own versions of functions such as `add` and `mult`. It is worth noting that this second form consumes significantly more space than the former.

v. If we want to define natural numbers from scratch, we also have to define addition and multiplication recursively. Both addition and multiplication are functions of two arguments. Using the alternative definition that we discussed in class, which of the two arguments can be used to apply the template for consuming a natural number?

- 6 points

  Since both arguments are recursively defined struct-lists, there is no *prima facie* reason that we could not apply our (recursive) template to either one. Since addition and multiplication are both commutative (e.g., $a * b = b * a$), we can indeed choose either one.

Problem 5. (Additional page for your solution)