COMP 210: Principles of Computing and Programming
First Exam

Solution Key

September 27th, 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	**20**	
2	**15**	
3	**25**	
4	**10**	
5	**30**	
Total	100	

Problem 1. (20 points) Given the following definition for the function `f`

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

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 (number? 'a) (= 'a 1)) (or false (/ 1 0))]
                [(or (symbol? 'b) (= a 2))   (/ a 2)]))
  (f 2)

= (cond [(and (number? 'a) (= 'a 1)) (or false (/ 1 0))]
        [(or (symbol? 'b) (= 2 2))   (/ 2 2)]))     //Substitution: 3 points

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

= (cond [false (or false (/ 1 0))]                  //(and false...): 3 points
        [(or (symbol? 'b) (= 2 2))   (/ 2 2)]))

= (cond [(or (symbol? 'b) (= 2 2))   (/ 2 2)]))     //cond [false ...: 3 points

= (cond [(or true (= 2 2))   (/ 2 2)]))             //symbol?: 2 points

= (cond [true (/ 2 2)])                             //(or true...): 3 points

= (/ 2 2)                                           //cond [true ...: 3 points

= 1                                                 //Correct answer: 2 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.
- Continuing to evaluate (or true (= 2 2)) to (or true true)

Problem 2. (15 points) Which of the following are *values*? Add YES or NO next to each one.

Solution: Interleaved with rest of problem statement. Each sub-problem was worth 1.5 points. Half-points were rounded up.

- `x`
  - No. A variable must be reduced (by looking up its value) before it becomes a value.
- `(f 5)`
  - No. A function call is not a value.
- `(cons 5)`
  - No. Cons takes two arguments, so, this is not even a syntactically valid expression.
- `empty`
  - Yes.
- `(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 (make-posn 5 6) empty)`
  - Yes.
- `(cons empty empty)`
  - Yes. This is an example value for a list of lists.
- `(cons empty false)`
  - No. Remember that cons actually takes its value to make sure that its second argument is a list.

Problem 3. (25 points) A *stack* is a data structure that has many applications in computer science, including automata theory, compilers, algorithms, and many others. Consider a stack of numbers: It can be empty, or it is a number and another stack.

i. Write down a data definition for a stack. Do not define any new structures in this problem.

ii. Write down three different examples of a stack

iii. Write down the template for a function that uses a stack

iv. Write down a function that counts the non-negative elements on the stack

Solution:

i. 6 points

```
; A stack is either
;   empty, or
;   (cons n s)
; where n is a number and n is a stack
```

ii. 6 points

```
empty
(cons 1 empty)
(cons 2 (cons 1 empty))
```

iii. 6 points

```
; f : stack -> ...
; (define (f x)
;   (cond [(empty? x) ...]
;         [else       ... (first x) ... (f (rest x)) ...]))
```

iv. 7 points

```
count : stack -> number
(define (count x)
  (cond [(empty? x) 0]
        [else       (+ (cond [(>= (first x) 0) 1]
                             [else             0])
                       (count (rest x)))]))
```

Common Problems:

- In (i), defining it using a struct.
- In (ii), improperly parenthesizing the examples.
- In (ii), misunderstanding the question and giving a free-response answer (not a common problem, but serious for the 2-3 who did it)
- In (iii), leaving off the "else" for the second cond test
- In (iii), leaving off the recursive invocation on the rest.
- In (iv), collapsing the nested cond into 3 outer cond branches.
- In (iv), mixing up the logic that tests for non-negativity ("$(>= 0n)$", for example)
- In (iv), incorrectly handing the recursive case (not testing for negativity, for example)

Problem 4. (10 points) Expand `'((1 2) () (a b) c)` into an expression that uses only `empty`, `cons`, symbols, and numbers.

Solution: As an intermediate step we can expand the above to:

```
(list (list 1 2)
      empty
      (list 'a 'b)
      'c)
```

From this we can get to the final answer:

```
(cons (cons 1 (cons 2 empty))          // Line 1    (2 points)
(cons empty                            // Line 2    (2 points)
(cons (cons 'a (cons 'b empty))        // Line 3    (2 points)
(cons 'c empty))))                     // Line 4    (2 points)
                                       //  parens   (1 point)
                                       //  'a 'b .. (1 point)
```

Common problems:

- General: Missing quotes on symbols. Unnecessary operations in result. Missing `cons`. Extra `cons`.
- Line 1: Improper parenthesization.
- Line 2: Replacing `empty` with `(cons empty empty)`.
- Line 4: Extra `empty`.

Problem 5. (30 points) You now know that natural numbers are a special subset of numbers, and that they can be defined as a recursive type. In fact, you now have pretty much all the tools needed to define all operations on natural numbers from scratch. So, **without** using any built in operations on numbers (like $+$, $*$, etc),

i. Write a data definition that defines the natural numbers.

ii. Write the template for a function that consumes a natural number.

iii. Write the contract and code for a function that adds two natural numbers.

iv. Write the code for a function that multiplies two natural numbers.

v. Write the code for a function that takes two natural numbers i and j, and computes the result of raising i to the jth power (i^j).

Solution:

i. 6 points

```
; A natural is either
;    'Zero, or
;    (make-num n)
; where n is a natural
(define-struct num (less-one))
```

ii. 6 points

```
; f : natural -> ...
; (define (f x)
;   (cond [(symbol? x) ...]
;         [else        ... (f (num-less-one x)) ...]))
```

iii. 6 points

```
; add : natural natural -> natural        ;; This is the contract
 (define (add x y)
   (cond [(symbol? x) y]
         [else        (make-num (add (num-less-one x) y))]))
```

iv. 6 points

```
; mult : natural natural -> natural       ;; (Note required here)
 (define (mult x y)
   (cond [(symbol? x) 'Zero]
         [else        (add y (mult (num-less-one x) y))]))
```

v. 6 points

```
; pow : natural natural -> natural        ;; (Recursion on 2nd)
 (define (pow x y)
   (cond [(symbol? y) (make-num 'Zero)]
         [else        (mult x (pow x (num-less-one y)))]))
```

Common Problems:

- Inconsistency between the data definition and the template (interchanging "rest" with "sub1", for example)
- In (ii), not recurring in the recursive case
- In (ii), using "add1" instead of "sub1" to recur
- In (ii), mixing up lists with nats (including both "first" and "rest" calls, for example)
- In (iii)-(v), getting the base case wrong
- In (iii)-(v), getting the math in the recursive case wrong
- In (iii)-(v), adding unnecessary base cases (like both i^0 and i^1), which breaks the template.