CMSC 15100, Fall 2005

Midterm Exam 2 Section 01

Name ____

1a	(from 4)	
1b	(from 5)	
1c	(from 10)	
2a	(from 4)	
2b	(from 4)	
2c	(from 4)	
2d	(from 4)	
3a	(from 5)	
3b	(from 20)	
4a	(from 5)	
4b	(from 10)	
4c	(from 10)	
4d	(from 5)	
5	(from 10)	
Total	(from 100)	

Rule: You may use any function assigned as homework here without re-defining it. You may also use any function on this test in any other function.

CMSC 15100, Fall 2005

Midterm Exam 2 Section 02

Name ____

1a	(from 4)	
1b	(from 5)	
1c	(from 10)	
2a	(from 4)	
2b	(from 4)	
2c	(from 4)	
2d	(from 4)	
3a	(from 5)	
3b	(from 20)	
4a	(from 5)	
4b	(from 10)	
4c	(from 10)	
4d	(from 5)	
5	(from 10)	
Total	(from 100)	

Rule: You may use any function assigned as homework here without re-defining it. You may also use any function on this test in any other function.

1 [19 total points]. A (*tree-of X*) can be defined as follows:

```
;; A (tree-of X) is either:
;; - (make-leaf X)
;; - (make-branch (tree-list-of X))
;; A (tree-list-of X) is:
;; - (listof (tree-of X))
(define-struct leaf (item))
(define-struct branch (subtrees))
```

For example, the following is a (*tree-of num*):

```
(define a-tree (make-branch (list (make-leaf 1)
                                  (make-branch (list (make-leaf 2) (make-leaf 3)))
                                  (make-leaf 4))))
```

1a [4 points]. Write an example (*tree-of boolean*) and an example (*tree-of symbol*). Include at least two branch structures in each.

Solution

```
(define tree-2 (make-branch (list (make-branch (list (make-leaf true))))))
(define tree-3 (make-branch (list (make-leaf 'hello)
                                  (make-branch (list (make-leaf 'world)
                                                     (make-leaf '!))))))
```

1b [5 points]. Write a template for functions that process (*tree-of X*) values.
Solution

```
(define (fft/tree tox)
  (cond
    ((leaf? tox) ··· (leaf-item tox) ···)
    ((branch? tox)
     ··· (fft/list (branch-subtrees tox)) ···)))
(define (fft/list lot)
  (cond
    ((empty? lot) ···)
    (else
     ··· (fft/tree (first lot)) ···
     ··· (fft/list (rest lot)) ···)))
```

1c [10 points]. Develop the function *fringe : (tree-of X)* $\rightarrow$ *(listof X)*, which takes a tree and produces the list of elements contained in its leaf nodes. For instance, (*fringe a-tree*) = (*list* 1 2 3 4).

Solution

```
(define (fringe tox)
  (cond
    ((leaf? tox) (list (leaf-item tox)))
    ((branch? tox)
     (fringe/subtrees (branch-subtrees tox)))))
(define (fringe/subtrees lot)
  (cond
    ((empty? lot) empty)
    (else
     (append (fringe (first lot))
             (fringe/subtrees (rest lot))))))
```

2 [16 total points]. For each of these contracts and purpose statements, write a function that would implement it by *calling the appropriate helper functions* with the appropriate arguments. You may use *filter*, *map*, and *ormap* (you have not seen *ormap* in class). None of the answers to these questions should be recursive functions themselves. You do not need to write test cases.

```
;; filter : (X → boolean) (listof X) → (listof X)
;; returns the elements of lox for which (f x) returns true
(define (filter f lox)
  (cond
    [(empty? lox) empty]
    [else
     (cond
       [(f (first lox))
        (cons (first lox) (filter f (rest lox)))]
       [else (filter f (rest lox))])]))

;; map : (X → Y) (listof X) → (listof Y)
;; constructs a list by applying f to each element in lox
(define (map f lox)
  (cond
    [(empty? lox) empty]
    [else
     (cons (f (first lox)) (map f (rest lox)))]))

;; ormap : (X → boolean) (listof X) → boolean
;; determines if the given function returns true for at least one element in the given list
(define (ormap f lox)
  (cond
    [(empty? lox) false]
    [else
     (or (f (first lox)) (ormap f (rest lox)))]))
```

2a [4 points]. *add3-to-all : (listof number) → (listof number)* Returns a list of each number in the input list with 3 added to it. E.g., (*add3-to-all* (*list* 0 1 2 3)) = (*list* 3 4 5 6).

Solution

```
(define (add3-to-all lon)
  (map (lambda (x) (+ x 3)) lon))
```

2b [4 points]. *any-even? : (listof number) → boolean*. Returns true if the given list of numbers contains an even number.

Solution

```
(define (any-even? lon)
  (ormap even? lon))
```

2c [4 points]. *zeroes? : (listof (listof num)) → boolean*. Returns true if and only if there there is some input list that contains a zero. E.g.,

```
(zeroes? (list empty (list 1))) = false
(zeroes? (list (list 1 0 2) (list 0))) = true
```

Solution

```
(define (zeroes? lolon)
  (ormap
   (lambda (lon) (ormap zero? lon))
   lolon))
```

2e [4 points]. *remove-zeroes* (*listof* (*listof num*)) → (*listof* (*listof num*)). Eliminates all zeroes from the input lists. E.g.,

```
(remove-zeroes (list empty (list 1))) = (list empty (list 1)))
(remove-zeroes (list (list 1 0 2) (list 0))) = (list (list 1 2) empty))
```

Solution

```
(define (remove-zeroes lolon)
  (map
   (lambda (lon)
     (filter (lambda (n) (not (zero? n))) lon))
   lolon))
```

3 [25 total points]. Some graphs have the interesting property that they contain at least one node with exactly as many neighbors as the average number of neighbors of all nodes in the graph (graphs with no nodes do not have this property). Recall from class:

```
;; a (graph-of X) is:
;; (make-graph (listof X) (X → (listof X)) (X X → boolean))
(define-struct graph (nodes neighbors node-equal?))
```

3a [5 points]. Define two graphs, *graph1* and *graph2*. The first must have this property, and the second must not.

Solution

```
(define test1 (make-graph (list 'a 'b 'c) (lambda (x) (list 'a 'b 'c)) symbol=?))
(define test2 (make-graph (list 'a 'b 'c)
                          (lambda (x)
                            (cond
                              ((symbol=? 'a x) '())
                              (else (list 'a 'b 'c))))
                          symbol=?))
```

3b [20 points]. Develop the function *average? :* (*graph-of X*) → *boolean* which determines whether a given graph has this property. In your solution, you may use the function *avg* which computes the average of a list of numbers:

```
;; avg : (listof num) → num
(define (avg lon)
  (/ (foldl + 0 lon) (length lon)))
```

Solution

```
(define (average? G)
  (local ((define neighbors (graph-neighbors G)))
    (ormap
     (lambda (node)
       (= (length (neighbors node)) (graph-average G)))
     (graph-nodes G))))
(define (graph-average G)
  (avg (map
        (lambda (n) (length ((graph-neighbors G) n)))
        (graph-nodes G))))
```

4 [30 total points]. A number can be naturally encoded as a *base-10-nat*, which is either a digit or another *base-10-nat* with an extra digit at the end. More formally:

```
;; a base-10-nat is either:
;; - a digit (i.e., a number from 0 to 9)
;; - (+ (* 10 n) d) where n is a base-10-nat and d is a digit

(define (digit? n) (< n 10))
```

The selectors for a multi-digit *base-10-nat* n are *all-but-last-digit* and *last-digit*, which can be defined as follows:

```
;; all-but-last-digit : base-10-nat → base-10-nat
(define (all-but-last-digit n)
  (quotient n 10))

;; last-digit : base-10-nat → digit
(define (last-digit n)
  (remainder n 10))
```

For example, the number 123 is a *base-10-nat* since it can be written as (+ (∗ 10 12) 3) and 12 is a *base-10-nat* and 3 is a digit. Furthermore:

```
(all-but-last-digit 123) = 12
(last-digit 123) = 3
```

4a [5 points]. Write a template for functions that process *base-10-nats*.
Solution

```
(define (fun-for-b10n b10n)
  (cond
    ((digit? b10n) ···)
    (else
     ··· (fun-for-b10n (all-but-last-digit b10n)) ···
     ··· (last-digit b10n) ···)))
```

4b [10 points]. Develop the function *number→digits : base-10-nat* → (*listof digit*). This function is the inverse of the function *digits→number* you developed for homework: *digits→number* produced a number from a given sequence of digits, and *number→digits* produces the sequence of digits represented by the given number. For example, (*number→digits* 3) = (*list* 3) and (*number→digits* 123) = (*list* 1 2 3).

Do not use an accumulator.

Solution

```
(define (number→digits b10n)
  (cond
    ((digit? b10n) (list b10n))
    (else
     (append (number→digits (all-but-last-digit b10n))
             (list (last-digit b10n))))))
```

4c [10 points]. Your *number→digits* function should be a good candidate for being rewritten in accumulator style. Concisely describe a useful accumulator for this function, state its initial value, and state an accumulator invariant for it. Recall that an accumulator invariant is a logical statement relating three things: the input to the original function, the input to the accumulator version of the function, and the accumulator.

Solution

An accumulator could hold the list of digits representing the conversion of the suffix of the number being converted into a list. It would start with empty. Accumulator invariant: if n_1 is the original argument to *number→digits*, n is the argument to the accumulator function, and *acc* is the accumulator, then the digits of n_1 are the digits of n followed by *acc*.

4d [5 points]. Write a revised version of *number→digits* that maintains and exploits an accumulator.
Solution

```
(define (number→digits n)
  (local ((define (n→d n acc)
            (cond
              ((digit? n) (cons n acc))
              (else (n→d
                      (all-but-last-digit n)
                      (cons (last-digit n) acc))))))
    (n→d n empty)))
```

5 [10 points]. A *number palindrome* is a number that is the same when read from left to right or from right to left. It is possible to associate a palindrome with a positive number n using the following rule:

- If n is a palindrome, it is associated with itself.
- If n is not a palindrome, then it is associated with whatever palindrome is associated with the number formed by adding n to the number formed by reversing the digits of n.

For instance, 14741 is associated with itself; 23 is associated with 55 because $23 + 32 = 55$ and 55 is a palindrome; and 87 is associated with 4884 because $87 + 78 = 165$ and $165 + 561 = 726$ and $726 + 627 = 1353$ and $1353 + 3531 = 4884$ and 4884 is a palindrome.[1]

Develop the function *find-palindrome : number → number*, which determines the palindrome associated with the given number. For instance, (*find-palindrome* 14741) = 14741, (*find-palindrome* 23) = 55, and (*find-palindrome* 87) = 4884. You may find these two helper functions useful:

```
;; reverse-num : number → number
;; reverses the digits of a number
;; e.g., (reverse-num 123) = 321
(define (reverse-num n)
  (digits→number (reverse (number→digits n))))

;; palindrome? : number → boolean
;; determines if a number is a palindrome.
;; e.g., (palindrome? 123) = false, (palindrome? 12321) = true
(define (palindrome? n)
  (= n (reverse-num n)))
```

Solution

```
(define (find-palindrome n)
  (cond
    ((palindrome? n) n)
    (else (find-palindrome (+ n (reverse-num n))))))
```

[1]Curiously, no one knows whether 196 has any palindrome associated with it at all. Numbers that have no associated palindrome are called *Lychrel numbers*, and no one knows whether any Lychrel numbers actually exist. We *do* know that if 196 has an associated palindrome, that palindrome must be over 278,750,715 digits long!