CMSC 15100, Fall 2004
Section 1
Exam 2

Name _____

1	(from 20)	
2	(from 25)	
3	(from 25)	
4	(from 10)	
5	(from 20)	
total	(from 100)	

Recall *reverse* and *add-at-end*:

```
;; reverse : list-of-numbers → list-of-numbers
(define (reverse l)
  (cond
    [(empty? l) empty]
    [else (add-at-end (first l) (reverse (rest l)))]))

;; add-at-end : number list-of-numbers → list-of-numbers

(define (add-at-end ele l)
  (cond
    [(empty? l) (list ele)]
    [else (cons (first l) (add-at-end ele (rest l)))]))
```

Rewrite these functions, using *fold*:

```
;; fold : list-of-X Y (X Y → Y) → Y
(define (fold l base combine)
  (cond
    [(empty? l) base]
    [else (combine (first l) (fold (rest l) base combine))]))
```

For each use of *fold*, identify what X and Y from *fold*'s type are.

Solution

```
(define (reverse l) (fold l empty add-at-end))
(define (add-at-end ele l) (fold l (list ele) cons))
```

In both cases, X is number and Y is list-of-numbers.

Here is a data definition for a set of numbers. Unlike a list of numbers, a set of numbers should not contain any duplicated elements.

```
;; a set-of-numbers is either
;; - empty
;; - (cons number[n] set-of-numbers[l])
;; INVARIANT: the number n is not in the list-of-numbers l
```

Not all sets have unique representations. For example, the set of numbers $\{1, 3\}$ can be represented as either

```
(cons 1 (cons 3 empty))
```

or

```
(cons 3 (cons 1 empty))
```

These should be thought of as equivalent sets.

Develop three functions:

```
;; start : number → set-of-numbers
;; to build a new set of numbers that contains only n
(define (start n) ...)

;; extend : number set-of-numbers → set-of-numbers
;; to build a bigger set of numbers, extending son.
(define (extend n son) ...)

;; test : number set-of-numbers → boolean
;; to determine if n is in son.
(define (test n son) ...)
```

Solution

```
;; start : number → set-of-numbers
;; to build a new set of numbers that contains only n
(define (start n) (list n))

;; extend : number set-of-numbers → set-of-numbers
;; to build a bigger set of numbers, extending son.
(define (extend n son)
  (cond
    [(test n son) son]
    [else (cons n son)]))

;; test : number set-of-numbers → boolean
;; to determine if n is in son.
(define (test n son)
  (cond
    [(empty? son) false]
    [else (or (= n (first son))
              (test n (rest son)))]))
```

Here is another data definition for a set of numbers:

```
;; a set of numbers is a function:
;; number → boolean
```

The intention is that applying the set to a number determines if the number is in the set. For example, this function:

```
(lambda (x) false)
```

represents the set with no numbers and this function:

```
(lambda (x) (or (= x 2) (= x 1)))
```

represents the set that contains only the numbers 1 and 2.

Develop the same three functions from the previous page, but using the new data definition:

```
;; start : number → set-of-numbers
;; to build a new set of numbers that contains only n
(define (start n) ...)

;; extend : number set-of-numbers → set-of-numbers
;; to build a bigger set of numbers, extending son.
(define (extend n son) ...)

;; test : number set-of-numbers → boolean
;; to determine if n is in son.
(define (test n son) ...)
```

Solution

```
;; start : number → set-of-numbers
;; to build a new set of numbers that contains only n
(define (start n) (lambda (x) (= x n)))

;; extend : number set-of-numbers → set-of-numbers
;; to build a bigger set of numbers, extending son.
(define (extend n son)
  (lambda (y)
    (or (= n y)
        (son y))))

;; test : number set-of-numbers → boolean
;; to determine if n is in son.
(define (test n son)
  (son n))
```

```
;; merge-sort : list-of-numbers → list-of-numbers
(define (merge-sort l)
  (cond
    [(empty? l) empty]
    [else
     (merge (merge-sort (evens l))
            (merge-sort (odds l)))]))

;; merge : list-of-numbers list-of-numbers → list-of-numbers
(define (merge l1 l2)
  (cond
    [(empty? l1) l2]
    [(empty? l2) l1]
    [else
     (cond
       [(<= (first l1) (first l2))
        (cons (first l1) (merge (rest l1) l2))]
       [else (cons (first l2) (merge l1 (rest l2)))])]))

;; evens : non-empty-list-of-numbers → list-of-numbers
;; to extract alternating elements of l, skipping the first one.
(define (evens l)
  (cond
    [(empty? (rest l)) empty]
    [else (odds (rest l))]))

;; odds : non-empty-list-of-numbers → list-of-numbers
;; to extract alternating elements of l, starting with the first one.
(define (odds l)
  (cond
    [(empty? (rest l)) l]
    [else (cons (first l) (evens (rest l)))]))

;; (some) examples
(evens (list 1 2 3 4)) = (list 2 4)
(odds (list 1 2 3 4)) = (list 1 3)
```

Is the function *merge-sort* generative or structurally recursive?

Solution
Generative

Is the function *merge* generative or structurally recursive?

Solution
Structural

The *merge-sort* function on the previous page does not terminate for all lists of numbers. Identify an input for which it fails to terminate. Provide a fix so that it will terminate for all lists of numbers.

Hint: try some (small) hand evaluations.

Solution

merge-sort doesn't make progress for a list of numbers that only has one number in it. For example:

```
    (merge-sort (list 1))
=   (merge (merge-sort empty) (merge-sort (list 1)))
=   (merge (merge-sort empty) (merge (merge-sort empty) (merge-sort (list 1))))
=   ...
```

To fix, add a case for a singleton list to *merge-sort*.

```
;; merge-sort : list-of-numbers → list-of-numbers
(define (merge-sort l)
  (cond
    [(empty? l) empty]
    [(empty? (rest l)) l]
    [else
     (merge (merge-sort (evens l))
            (merge-sort (odds l)))]))
```