CMSC 15100, Fall 2004
Section 1
Final Exam

Name ____ (1pt)

Do not repeat steps of the design recipe that are given in the problem.

1	(from 11)	
2	(from 11)	
3	(from 11)	
4	(from 11)	
5	(from 11)	
6	(from 11)	
7	(from 11)	
8	(from 11)	
9	(from 11)	
total	(from 100)	

```
;; a number-tree is either:
;; number
;; (make-node number-tree number-tree)
(define-struct node (left right))
```

Design the function *same-shape?*. It consumes two number trees and returns a boolean indicating if the trees have the same shape, regardless of the numbers in the tree.

```
;; examples
(equal? (same-shape? 1 2) true)
(equal? (same-shape? (make-node 1 2) (make-node 3 4)) true)
(equal? (same-shape? (make-node 1 2) 5) false)
(equal? (same-shape? (make-node 1 (make-node 2 3))
                     (make-node (make-node 1 2) 3))
        false)
```

Solution

```
;; same-shape? : number-tree number-tree → boolean
(define (same-shape? t1 t2)
  (cond
    [(and (number? t1) (number? t2)) true]
    [(and (node? t1) (node? t2))
     (and (same-shape? (node-left t1) (node-left t2))
          (same-shape? (node-right t2) (node-right t2)))]
    [else false]))
```

Design the function *same-fringe?*. It consumes two number trees and returns true if the trees have the same numbers, regardless of the shape.

```
;; examples
(equal? (same-fringe? 1 2) false)
(equal? (same-fringe? 1 1) true)
(equal? (same-fringe? (make-node 1 2) (make-node 3 4)) false)
(equal? (same-fringe? (make-node 1 2) (make-node 1 2)) true)
(equal? (same-fringe? (make-node 1 2) 5) false)
(equal? (same-fringe? (make-node 1 (make-node 2 3))
                      (make-node (make-node 1 2) 3))
        true)
```

Solution

```
;; same-fringe? : number-tree number-tree → boolean
(define (same-fringe? t1 t2)
  (same-list? (to-list t1)
              (to-list t2)))

;; to-list : number-tree → (listof numbers)
(define (to-list t)
  (cond
    [(number? t) (list t)]
    [else (append (to-list (node-left t))
                  (to-list (node-right t)))]))

;; same-list? : list-of-numbers list-of-numbers → boolean
(define (same-list? l1 l2)
  (cond
    [(and (empty? l1) (empty? l2)) true]
    [else (and (= (first l1) (first l2))
               (same-list? (rest l1) (rest l2)))]))
```

Rewrite the following function into accumulator style. Identify the accumulator invariant.

```
;; all-true? : (listof boolean) → boolean
(define (all-true? lob)
  (cond
    [(empty? lob) true]
    [else (and (first lob)
               (all-true? (rest lob)))]))
```

Solution

```
(define (all-true? lob)
  (all-true?/a lob true))

;; all-true?/a : (listof boolean) boolean → boolean
;; the accumulator indicates if we have not seen any falses so far
(define (all-true?/a lob a)
  (cond
    [(empty? lob) a]
    [else (all-true? (rest lob)
                     (and (first lob) a))]))
```

Rewrite the following accumulator function to use foldl.

```
;; even-trues? : (listof booleans) → boolean
;; to determine if lob contains an even number of trues.
(define (even-trues? lob)
  (even-trues?/a lob true))

;; even-trues?/a : (listof booleans) boolean → boolean
;; the accumulator is true if there were an even number of
;; trues seen so far and false if there were an odd number of trues
;; seen so far.
(define (even-trues?/a lob a)
  (cond
    [(empty? lob) a]
    [else (even-trues?/a (rest lob)
                         (cond
                           [(first lob) (not a)]
                           [else a]))]))

;; foldl : (listof X) Y (X Y → Y) → Y
(define (foldl l a combine)
  (cond
    [(empty? l) a]
    [else (foldl (rest l) (combine (first l) a) combine)]))
```

Solution

```
(define (even-trues? l)
  (foldl l true (lambda (x y) (not (boolean=? x y)))))
```

Use hand-evaluation (ala Intermediate Scheme with lambda) to find the value of the following expression:

```
((lambda (f) (f (f 2)))
 (lambda (x) (* x 2)))
```

Do not skip any steps.

Solution

```
((lambda (f) (f (f 2)))
 (lambda (x) (* x 2)))
=
((lambda (x) (* x 2)) ((lambda (x) (* x 2)) 2))
=
((lambda (x) (* x 2)) (* 2 2))
=
((lambda (x) (* x 2)) 4)
=
(* 4 2)
=
8
```

Consider this data definition for a pipe:

```
;; a pipe is either:
;; - 'faucet
;; - (make-copper pipe)
;; - (make-iron pipe)
;; - (make-joint pipe pipe)

(define-struct copper (next))
(define-struct iron (next))
(define-struct joint (left right))
```

Pipes are terminated by faucets. The *make-copper* constructor builds a slightly bigger pipe by adding a copper length to an existing pipe. Similarly, *make-iron* adds an iron length to an existing pipe. The *make-joint* constructor combines two sections of pipe.

As any homeowner soon learns, plumbing is a black art. For example, if you have copper and iron piping right next to each other in the same hot-water line, the pipe will leak (the copper and iron expand and contract at different rates). Pipe joints, however, are specially designed to be able to connect to either copper or iron pipes safely (without causing any leaks).

To help homeowners everywhere, write a function that takes a description of a hot water pipe and determines if it would leak. Here are some example uses of that function:

```
(equal? (will-leak? 'faucet) false)
(equal? (will-leak? (make-copper 'faucet)) false)
(equal? (will-leak? (make-iron 'faucet)) false)
(equal? (will-leak? (make-copper (make-iron 'faucet))) true)
(equal? (will-leak? (make-iron (make-copper 'faucet))) true)
(equal? (will-leak? (make-iron (make-joint (make-copper 'faucet)
                                           (make-copper 'faucet))))
        false)
(equal? (will-leak? (make-joint (make-copper 'faucet)
                                (make-iron 'faucet))))
        false)
```

[scratch space, if neccessary]
Solution

```
;; will-leak? : pipe → boolean
(define (will-leak? pipe)
  (will-leak?/a pipe false))

;; will-leak?/a : pipe prev-section or false → boolean
(define (will-leak?/a pipe prev-section)
  (cond
    [(symbol? pipe) false]
    [(copper? pipe)
     (cond
       [(iron? prev-section) true]
       [else (will-leak?/a (copper-next pipe) pipe)])]
    [(iron? pipe)
     (cond
       [(copper? prev-section) true]
       [else (will-leak?/a (iron-next pipe) pipe)])]
    [(joint? pipe)
     (or (will-leak?/a (joint-left pipe) pipe)
         (will-leak?/a (joint-right pipe) pipe))]))
```

Design a function that takes in an integer bigger than or equal to 1 and produces a pipe.

```
;; fit-together : number → pipe
(define (fit-together faucet-count) ...)
```

The pipe should contain *faucet-count* faucets and the minimum number of joints (and no copper or iron pipes).

```
(equal? (fit-together 1)
        'faucet)
(equal? (fit-together 2)
        (make-joint 'faucet 'faucet))
(equal? (fit-together 3)
        (make-joint 'faucet (make-joint 'faucet 'faucet)))
(equal? (fit-together 4)
        (make-joint (make-joint 'faucet 'faucet) (make-joint 'faucet 'faucet)))
(equal? (fit-together 8)
        (make-joint
         (make-joint (make-joint 'faucet 'faucet)
                     (make-joint 'faucet 'faucet))
         (make-joint (make-joint 'faucet 'faucet)
                     (make-joint 'faucet 'faucet))))
```

Hint: if the number of faucets is even, the best arrangement is a single joint with half of the faucets on each side.

Solution

```
(define (fit-together faucets)
  (cond
    [(= 1 faucets) 'faucet]
    [(even? faucets)
     (make-joint (fit-together (quotient faucets 2))
                 (fit-together (quotient faucets 2)))]
    [(odd? faucets)
     (make-joint (fit-together (quotient faucets 2))
                 (fit-together (+ (quotient faucets 2) 1)))]))
```

```
;; an exp is either
;; - number
;; - (make-op exp (number number → number) exp)
(define-struct op (left fun right))
```

Translate the following Scheme expressions into *exp*s. Be careful about the order of the arguments to *make-op*.

```
(+ 1 2)

(+ (* 2 3) 4)

(* (* 3 2) (+ 2 3))
```

Solution

```
(make-op 1 + 2)
(make-op (make-op 2 * 3) + 4)
(make-op (make-op 3 * 2) * (make-op 2 + 3))
```

Write a function that accepts an *exp* and produces a number by evaluating the *exp*.
Solution

```
;; eval : paren-less-exp -¿ number
(define (eval exp)
  (cond
    [(number? exp) exp]
    [(op? exp) ((op-fun exp)
                (eval (op-left exp))
                (eval (op-right exp)))]))
```