Question 1 (5 points):

Write a function `multicompose` that takes as its argument a list of any length, whose elements are one-argument functions. It should return a one-argument function that carries out the composition of all of the given functions, like this:

```
> (define third (multicompose (list first bf bf)))
THIRD
> (third '(the long and winding road))
AND

> ((multicompose (list square square 1+)) 2)
81

> ((multicompose (list square 1+ sqrt 1+)) 8)
16
```

Your solution must use functional programming style — no mutation!

Question 2 (5 points):

Imagine that you are working in a version of Scheme in which the only numbers available are integers. You want to be able to represent amounts of money, such as $49.95, but you can't use the obvious representation `49.95` because that isn't an integer.

You decide to create an abstract data type called a *price* that has two components, called `dollars` and `cents`. You write these constructors and selectors:

```
(define (make-price d c) (+ (* d 100) c))
(define (dollars p) (quotient p 100))
(define (cents p) (remainder p 100))
```

(a) Write the procedure `+price` that takes two prices as arguments and returns their sum, as a new price. **Respect the data abstraction.**

(b) Now we want to change the internal representation so that instead of representing a price as a single number of cents, we represent it *internally* as a list of two numbers, one for the number of dollars and one for the number of leftover cents. Rewrite the constructors and selectors as needed. *Don't change the interface to procedures that use the abstraction.*

Question 3 (5 points):

Here is a partial implementation of a telephone system, modeled using objects. Each telephone is an object, and the telephone company's switching equipment is another kind of object.

When someone wants to place a call, they `ask` their telephone to `call` the number of the other person's phone:

```
(ask my-phone 'call 6428311)
```

The telephone asks the phone company to `connect` itself to that number. The result from this request can be one of three things:

- The word `NOT-IN-SERVICE` if there is no such telephone number.
- The word `BUSY` if the desired telephone is in use already.
- The word `CONNECTED` if the connection succeeds.

In the third of these cases, the calling phone should note the fact that it's now in use, by setting its `BUSY?` variable true.

The phone company must handle the `connect` request by finding the desired number in its `phones` variable, which is an association list of numbers and phones. If the phone exists, the phone company asks it if it's busy. If the phone isn't busy, the connection succeeds. The phone company must ask the called telephone to `ring`, and must also remember all current connections by maintaining a list of caller-callee pairs in its `connections` variable.

```
(define-class (phone num company)
  (instance-vars (busy? #f))
  (method (call other-num)
    ----YOU WILL WRITE THIS PART----)
  (method (ring)
    (set! busy #t))
  (method (hangup)
    (ask company 'disconnect self)
    (set! busy #f))
  (method (disconnect)
    (set! busy #f)))
```

This question continues on the next page.

Your name _____ login cs61a–_____

Question 3 continued.

```
(define-class (telephone-company)
  (instance-vars (phones '())
                 (next-number 6421000)
                 (connections '()))
  (method (new-phone)
    (set! next-number (1+ next-number))
    (set! phones (cons (cons next-number
                             (instantiate phone next-number self))
                       phones))
    (cdar phones))
  (method (connect caller num)
    ----YOU WILL WRITE THIS PART----)
  (method (disconnect phone)
    (let ((connection (find-other phone connections)))
      (if (not connection)
          #f
          (let ((other (if (eq? phone (car connection))
                           (cdr connection)
                           (car connection))))
            (ask other 'disconnect)
            (set! connections (remove connection connections)))))))
```

Here is the procedure `find-other`, used in the `disconnect` method for telephone companies. It is similar to `assq` except that it looks for the argument value as either the car or the cdr of each pair.

```
(define (find-other key alist)
  (cond ((null? alist) #f)
        ((eq? key (caar alist)) (car alist))
        ((eq? key (cdar alist)) (car alist))
        (else (find-other key (cdr alist)))))
```

(a) Write the `connect` method for telephone company objects.

(b) Write the `call` method for telephones.

Question 4 (5 points):

Ben Bitdiddle wants to create an infinite stream of random digits. He says

```
(define randoms (cons-stream (random 10) randoms))
```

(a) What's wrong with this? Describe the result that Ben gets.

(b) Fix it: Write the necessary code to accomplish what Ben wants.

Question 5 (5 points):

Write a set of rules and/or assertions in the query system to implement an `odd-length` relation like this:

```
QUERY==> (odd-length (she loves you))
(ODD-LENGTH (SHE LOVES YOU))
DONE

QUERY==> (odd-length (day tripper))
DONE
```

Queries using `odd-length` should succeed for lists containing an odd number of elements, and should fail for lists containing an even number of elements.

Do not use `lisp-value`!

Question 6 (5 points):

We'd like to add a *package* feature to the metacircular evaluator. This feature allows several programmers working together on a large project to avoid name conflicts by collecting the work of each programmer separately, and allowing other programmers to use those definitions only by an explicit request. Here's an example. Programmer A writes this:

```
(define sort-package
  (let ()                    ;; This just makes an empty frame!
    (define (sort stuff)
      (if (null? stuff)
          '()
          (insert (car stuff)
                  (sort (cdr stuff)))))
    (define (insert thing others)
      (cond ((null? others) (cons thing '()))
            ((< thing (car others)) (cons thing others))
            (else (cons (car others) (insert thing (cdr others))))))
    (define (merge a b)
      (cond ((null? a) b)
            ((null? b) a)
            ((< (car a) (car b))
             (cons (car a) (merge (cdr a) b)))
            (else (cons (car b) (merge a (cdr b))))))
    (export sort merge)))
```

This defines `sort-package` as a collection from which the procedures `sort` and `merge` are available. To use something from the package, another programmer must name it with a symbol consisting of the package name, a colon, and the exported name, like this:

```
(sort-package:sort '(4 27 8 19 6))
```

Notice that the procedure `insert` was defined inside the package, but is not exported, so you can't say `sort-package:insert` to use it. This procedure can only be used within

the package itself.

Your task is to modify the metacircular evaluator to allow this style of programming. You must invent the `export` feature, which takes any number of symbols and returns an environment frame (not a complete environment, just the one frame) with those symbols and their current binding values. Then you must make the evaluator understand symbols like `sort-package:sort`. When such a symbol is evaluated, the evaluator must first find the package (by evaluating that part of the name), then look up the specific desired symbol within the frame that represents the package.

You are given procedures `before-colon` and `after-colon` that take a symbol as argument and return the parts of the symbol on either side of the colon. If the argument symbol has no colon, then `before-colon` returns `#f`, and `after-colon` returns the entire argument symbol.

(a) Is `export` primarily a change to `eval` or to `apply`?

(b) Is the new symbol notation primarily a change to `eval` or to `apply`?

(c) What specific procedure(s) will you change for `export`?

(d) What specific procedure(s) will you change for the new symbol notation?

(e) Make the changes on the following pages. If you define any entirely new procedures, write them on the (blank) back page of the exam.

This question continues on the following page.

Your name _____ login cs61a–_____

Question 6 continued:

Here are the promised procedures for dissecting symbols:

```
(define (before-colon sym)
  (define (help left right)
    (cond ((empty? right) #f)
          ((equal? (first right) ':) left)
          (else (help (word left (first right)) (butfirst right)))))
  (help "" sym))

(define (after-colon sym)
  (define (help right)
    (cond ((empty? right) sym)
          ((equal? (first right) ':) (butfirst right))
          (else (help (butfirst right)))))
  (help sym))
```

Here and on the following pages are possibly relevant procedures from the metacircular evaluator:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence (procedure-body procedure)
                        (extend-environment
                         (parameters procedure)
                         arguments
                         (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

This question continues on the following page.

Question 6 continued:

```
(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (eval (first-operand exps) env)
                    (list-of-values (rest-operands exps)
                                    env)))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (extend-environment variables values base-env)
  (adjoin-frame (make-frame variables values) base-env))

(define (adjoin-frame frame env) (cons frame env))

(define (make-frame variables values)
  (cond ((and (null? variables) (null? values)) '())
        ((null? variables)
         (error "Too many values supplied" values))
        ((null? values)
         (error "Too few values supplied" variables))
        (else
         (cons (make-binding (car variables) (car values))
               (make-frame (cdr variables) (cdr values))))))

(define (make-procedure lambda-exp env)
       (list 'procedure lambda-exp env))

(define (make-binding variable value)
  (cons variable value))

(define (binding-variable binding)
  (car binding))

(define (binding-value binding)
  (cdr binding))

(define (set-binding-value! binding value)
  (set-cdr! binding value))
```

This question continues on the following page.

Your name _____ login cs61a–_____

Question 6 continued:

```
(define (lookup-variable-value var env)
  (let ((b (binding-in-env var env)))
    (if (found-binding? b)
        (binding-value b)
        (error "Unbound variable" var))))

(define (binding-in-env var env)
  (if (no-more-frames? env)
      no-binding
      (let ((b (binding-in-frame var (first-frame env))))
        (if (found-binding? b)
            b
            (binding-in-env var (rest-frames env))))))

(define (extend-environment variables values base-env)
  (adjoin-frame (make-frame variables values) base-env))

(define (set-variable-value! var val env)
  (let ((b (binding-in-env var env)))
    (if (found-binding? b)
        (set-binding-value! b val)
        (error "Unbound variable" var))))

(define (define-variable! var val env)
  (let ((b (binding-in-frame var (first-frame env))))
    (if (found-binding? b)
        (set-binding-value! b val)
        (set-first-frame!
          env
          (adjoin-binding (make-binding var val)
                          (first-frame env))))))

(define (eval-assignment exp env)
  (let ((new-value (mini-eval (assignment-value exp) env)))
    (set-variable-value! (assignment-variable exp)
                         new-value
                         env)
    new-value))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (mini-eval (definition-value exp) env)
                    env)
  (definition-variable exp))
```