Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for one side of one 8.5x11in piece of paper.
- **Please stop promptly at 10:20.**
- You can rip apart the pages, but please staple them back together before you leave.
- There are **100 points** total, distributed among 7 questions (most with multiple parts). (Six questions are worth 15 points. One question is worth 10 points.)
- When writing code, style matters, but don’t worry about indentation.

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit.**
- The questions are not necessarily in order of difficulty. **Skip around.**
- If you have questions, ask.
- Relax. You are here to learn.
1. (15 points)

(a) Write a function fold in Scheme that is like the fold function we studied in ML. Recall fold takes 3 arguments: a function, an initial-result, and a list. The function is applied to each element of the list and the “current-result” to produce the next “current-result.” Have your Scheme fold function take 3 arguments (do not use currying).

(b) Write a Scheme function largest-pos that takes a list and returns the largest positive number in the list (or 0 if the list contains no positive numbers). Use fold and no other use of recursion. Your function should work even when not every element of the list is a number.
2. (10 points) Given the Scheme program below, what are x, y, and z bound to? Explain your answer by explaining what the function bound to f returns. Hint: x and z are bound to lists of numbers.

(define f
  (let ([y 0])
    (lambda ()
      (begin (set! y (+ y 1))
        (let ([x y])
          (lambda () x))))))
(define x (list ((f)) ((f)) ((f))))
(define y (f))
(define z (list (y) (y) (y)))
3. (15 points) Consider this Scheme code:

(define (foo1 x y)
  (if (= x 0)
    42
    (* x y y)))

(define-syntax foo2
  (syntax-rules ()
    [(foo2 x y)
      (if (= x 0)
        42
        (* x y y))])))

(a) Explain how uses of foo1 and foo2 could behave differently. Give an example and explain how foo1 and foo2 would behave differently for your example.

(b) Define a macro foo3 that always behaves equivalently to foo1 (even though it would be better style just to use foo1). Do not use foo1 or any other helper functions in your solution.
4. (15 points) This problem considers TTPL (for teeny tiny programming language), which is a lot like MUPL from homework 5. Like MUPL, it is embedded in Scheme via struct definitions. It has fewer constructs than MUPL and all functions must have names (whether or not they are recursive). Here are the definitions and one interpreter:

(define-struct var (string)) ;; a variable, e.g., (make-var "foo")
(define-struct int (num)) ;; a constant number, e.g., (make-int 17)
(define-struct add (e1 e2)) ;; add two expressions
(define-struct fun (name formal body)) ;; a recursive 1-argument function
(define-struct app (funexp actual)) ;; function application
(define-struct closure (fun env)) ;; closures (made at run-time)

(define (envlookup env str)
  (cond [(null? env) (error "unbound variable during evaluation" str)]
        [(equal? (caar env) str) (cdar env)]
        [#t (envlookup (cdr env) str)]))

(define (eval-prog p)
  (letrec
    ([f (lambda (env p) (cond [[(var? p) (envlookup env (var-string p))]
                               [(int? p) p]
                               [(add? p) (let ([v1 (f env (add-e1 p))]
                                              [v2 (f env (add-e2 p))])
                                (if (and (int? v1) (int? v2))
                                  (make-int (+ (int-num v1) (int-num v2)))
                                  (error "TTPL addition applied to non-number")))]
                               [[(fun? p) (make-closure p env)]
                                [(app? p) (let ([cl (f env (app-funexp p))]
                                              [arg (f env (app-actual p))]
                                              (if (closure? cl)
                                                (let* ([fn (closure-fun cl)]
                                                       [b1 (cons (fun-formal fn) arg)]
                                                       [b2 (cons (fun-name fn) cl)]
                                                       [new-env (cons b1 (cons b2 (closure-env cl))))]
                                                  (f new-env (fun-body fn)))
                                                (error "TTPL function call with non-function")))]
                               [[(closure? p) p]
                                [#t (error "bad TTPL expression")])))
    (f () p))

Now suppose we have a different interpreter eval-prog-other that is exactly like eval-prog except the line [new-env (cons b1 (cons b2 (closure-env cl)))] is instead [new-env (cons b2 (cons b1 (closure-env cl)))].

Write a TTPL program such that calling eval-prog with your program returns (make-int 4) but calling eval-prog-other with your program raises an error. **Explain your answer.**

(Put your answer on the next page. Sample solution uses all 5 kinds of TTPL source expressions including one function definition and one function application. It is "a little tricky" but not very long. Focus on what is different between the two interpreters.)
Name: ____________________________________________

This page intentionally blank.
5. (15 points) Recall that the Ruby Enumerable module provides methods that work assuming the class that includes Enumerable implements each correctly.

(a) Write Ruby code that adds a fold method to the Enumerable module. This fold method is similar to the fold function we studied in ML. It should take 2 arguments, an instance of class Proc (which recall has a method call) and an initial-result. The Proc is called with each “element of self” and the “current-result” to produce the next “current-result.”

(b) Define a top-level Ruby function largest_pos that takes an array and returns the largest positive number in the array. Use the fold method you defined above (remember, the Array class includes the Enumerable module). You will need to use lambda. Do not use any other methods of the Array class or any sort of loop. You may assume every element of the array is a number, and that your definition of fold works correctly.
6. (15 points) Consider these two simple Ruby classes:

```ruby
class C # line 1
  def m x
    print (x.foo + x.foo)
    42
  end
end

class D # line 2
  def m x
    print (x.foo + x.bar)
    42
  end
end
```

(a) Describe everything that class C’s `m` method assumes about its argument in order for a call to `m` not to raise an error.

(b) Describe everything that class D’s `m` method assumes about its argument in order for a call to `m` not to raise an error.

(c) Assume we add a type system for preventing message-not-understood errors to Ruby similar to what we discussed in lecture and that we have a policy of “all subclasses are subtypes” (so any instance of a subclass must be substitutable in place of any instance of a superclass). Which of the following should type-check? **Explain your answers.**

   i. Making C a subclass of D (i.e., adding `< D to line 1)
   ii. Making D a subclass of C (i.e., adding `< C to line 2)
7. (15 points) In a Ruby method body, a use of an identifier such as `x` (in general, any identifier) could refer either to a variable (a local variable or a method parameter) or to a method (as sugar for calling `self.x`). In the real Ruby language, if a method `x` is in scope and a local variable `x` is also in scope, the variable shadows the method. However, for this problem, assume instead it is a run-time error to evaluate the expression `x` when both a variable and method are in scope with the same name `x`.

Consider this static-checker for programs: For every class `C`, if `C` or any of its superclasses defines a method with some name (for example, `x`), then no method in `C` may have an argument or local variable with the same name (that is, `x`).

(a) Explain why the static-checker described above is *not sound* for preventing the run-time error described above.

(b) Explain why the static-checker described above is *not complete* for preventing the run-time error described above.