Name:_____

# CSE 341, Winter 2008, Final Examination
# 19 March 2008

## 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.

Name:_____

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.

**Solution:**

(a)
```
(define (fold f init lst)
   (if (null? lst)
       init
       (fold f (f init (car lst)) (cdr lst))))
```

(b)
```
(define (largest-pos lst)
   (fold (lambda (sofar next)
            (if (and (number? next)
                     (> next sofar))
                next
                sofar))
          0 lst))
```

Name:_____

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)))
```

**Solution:**
The program binds `x` to the list `(1 2 3)` and `z` to the list `(4 4 4)`. `y` is bound to a function that returns 4. The function bound to `f` returns a thunk that when called for the $n^{th}$ time returns a thunk that always returns $n$.

Name:_____

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.

**Solution:**

(a) The arguments to the function `foo1` are always evaluated exactly once because that is how function calls work in Scheme. For the macro `foo2`, the second argument will not be evaluated if `x` is 0 and will be evaluated twice if `x` is not 0. As an example, `(foo1 0 (/ 1 0))` would raise an error, but `(foo2 0 (/ 1 0))` would return 0. As another example, `(foo1 1 (begin (print "x") 3))` would print one `x` and return 9, but `(foo1 1 (begin (print "x") 3))` would print two `x`'s and return 9.

(b)
```
(define-syntax foo3
   (syntax-rules ()
     [(foo2 x y)
      (let ([x1 x]
            [y1 y])
        (if (x1 0)
            42
            (* x1 y1 y1)))]))
```

Name:_____

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.*

**Solution:**

```
(make-app (make-fun "f" "f" (make-add (make-var "f") (make-var "f")))
          (make-int 2))
```

In `eval-prog` when a function uses the same variable for recursion and for the argument, the argument shadows the function name, so in our example application we add 2 and 2 to produce 4. In `eval-prog-other` the function name shadows the argument, so the addition tries to add closures, which raises an error.

Name:_____

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.

   **Solution:**

   (a)
```
module Enumerable
  def fold(f,init)
    each {|x| init = f.call(init,x)}
    init
  end
end
```

   (b)
```
def largest_pos arr
  arr.fold(lambda {|sofar,elt| if elt < sofar : sofar else elt end}, 0)
end
```

Name:_____

6. (**15** points) Consider these two simple Ruby classes:

```
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)

**Solution:**

(a) It assumes `x` is an object that has a zero-argument method named `foo` that returns an object that has a `+` method that takes another object (also returned by `foo`) and returns something that can be printed.

(b) It assumes `x` is an object that has a zero-argument method named `foo` that returns an object that has a `+` method that takes another object and returns something that can be printed. It also assumes `x` has a zero-argument method named `bar` that returns an object appropriate for passing to the `+` method of the object returned from `foo`.

(c) i. This should type-check. With this subtyping `C` overrides method `m` but the method makes fewer assumptions about its argument (see part (a)), so this is fine contravariant subtyping. Note: One could argue that this should not type-check since `C` and `D` might make incompatible assumptions about the argument type of the `+` method in the object returned by `x.foo`. Solutions that explained such an argument clearly received full credit (though only 1–2 took this interpretatio of the situation).

ii. This should not type-check. With this subtyping `D` overrides method `m` but makes more assumptions about its arguments. So given an object of type `C` that was actually a `D` we might call `m` with an argument that has a `foo` method but not a `bar` method.

Name:_____

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.

   **Solution:**

   (a) It accepts some programs that can raise a run-time error. If a *subclass* of `C` extends `C` with a method `x`, then this new method is still in scope (due to dynamic dispatch), so we still must disallow variables named `x` in `C`'s methods. But the static-checker only checks `C` and its superclasses for what names must be avoided.

   (b) It rejects some programs that do not raise run-time errors. As a simple example, suppose the code in some method `m` is the only code that causes the static checker to reject a program. If the program never calls `m` when it runs, then we need not reject the program to avoid a run-time error.