Name:____

CSE341, Fall 2011, Final Examination
December 13, 2011

Please do not turn the page until the bell rings.

Rules:

- The exam is closed-book, closed-note, except for **both sides** of one 8.5x11in piece of paper.
- **Please stop promptly at 4:20.**
- You can rip apart the pages, but please staple them back together before you leave.
- There are **120 points** total, distributed **unevenly** among **8** questions (most with multiple parts).
- When writing code, style matters, but don't worry much 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.** Make sure you get to all the problems.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

1. (a) (**8** points) Write a Racket function `flip-if` that behaves as follows:
   - It takes two arguments, a two-argument function `f` and an association list (a list of pairs) `xs` and returns a list.
   - For each pair in `xs`, if `f` called with the pieces of the pair does not return false, then a pair with these pieces in reverse order is in the output. Else this pair is not in the output.
   - Even though the pieces of a pair in the output are in reverse order, the pairs in the output are in the same order as the pairs in `xs`.

   For example,

   ```
   (flip-if (lambda (x y) (< (- x y) 2))
            (list (cons 4 2) (cons 3 2) (cons 1 2) (cons 9 5)))
   ```

   evaluates to `'((2 . 3) (2 . 1))`.

   (b) (**5** points) Port your part (a) answer to SML to produce an ML function `flip_if` of type `(('a * 'b) -> bool) * (('a * 'b) list) -> (('b * 'a) list)`. For example, `flip-if ((fn (x,y) => x - y < 2), [(4,2),(3,2),(1,2),(9,5)])` evaluates to `[(2,3),(2,1)]`.

   (c) (**6** points) Show example calls to `flip-if` in Racket and `flip_if` in SML such that:
   - The Racket code runs without error and produces a non-empty list.
   - The SML code is a straightforward port of the Racket call, i.e., a function that does the same thing and a list with the same contents.
   - The SML code does not type-check.

   Explain in English why the SML call does not type-check.

Solution:

(a)
```
(define (flip-if f xs)
  (cond [(null? xs)
         null]
        [(f (caar xs) (cdar xs))
         (cons (cons (cdar xs) (caar xs))
               (flip-if f (cdr xs)))]
        [#t (flip-if f (cdr xs))]))
```

(b)
```
fun flip_if (f, xs) =
  case xs of
     [] => []
  | (k,v)::xs => if f(k,v)
                 then (v,k)::(flip_if(f,xs))
                 else flip_if(f,xs)
```

(c) There are several reasonable solutions; here are three approaches:

- Use an argument where pairs in the list do not all have the same type but the Racket function still works, for example:

  ```
  (flip-if (lambda (x y) (> x 0)) (list (cons 3 "hi") (cons 4 #t)))
  flip_if((fn (x,y) => x > 0), [(3,"hi"),(4,true)])
  ```

  The Racket call would produce `'(("hi" . 3)(#t . 4))`. It would not type-check in ML because all list elements must have the same type. The list `[(3,"hi"),(4,true)]` does not type-check because it has elements of type `int*string` and `int*bool`, not the same type.

- Use a function that does not type-check for some reason that does not affect execution, for example:

  ```
  (flip-if (lambda (x y) (if true x (+ 0 "hi"))) (list (cons 1 2)))
  flip_if((fn (x,y) => if true then x else "hi" + 0), [(1,2)]
  ```

  The ML code does not type-check because an argument to `+` is a string.
- Use a function that has multiple return types or a return type that is not a boolean as used, for example:

  ```
  (flip-if (lambda (x y) x) (list (cons 1 2)))
  flip_if((fn (x,y) => x), [(1,2)]
  ```

  The ML code does not type-check because the function will return an `int` where `flip_if` requires a `bool`, but in Racket any value can be used in a conditional.

Name:____

2. Recall we defined a stream to be a thunk that, when called, produces a pair of a value and another stream. Note the problems below are separate; the answer to one does not help answer another.

   (a) (**7** points) Write a Racket function `partA` that takes a stream and counts how many elements can be retrieved from the stream before encountering the element `#f`. If the first stream value is `#f`, the answer is 0, else if the next element is `#f`, the answer is 1, etc.

   (b) (**7** points) Write a Racket function `partB` that takes two streams and returns a stream. The n^{th} element of the output stream should be the n^{th} element of the first argument stream unless it is `#f` in which case the n^{th} element of the output stream should be the n^{th} element of the second argument stream (even if it is also `#f`).

   (c) (**3** points) *(Low-point total only because it is like a challenge problem)* Write a Racket function `partC` that takes a stream and returns a stream. The result should be like the argument except any `#f` values are skipped.

Solution:

(a)
```
(define (partA s)
   (let ([pr (s)])
      (if (car pr)
          (+ 1 (partA (cdr pr)))
          0)))
```

(b) Two reasonable solutions:

```
(define (partB s1 s2)
  (lambda ()
    (let ([pr1 (s1)]
          [pr2 (s2)])
      (cons (or (car pr1) (car pr2))
            (partB (cdr pr1) (cdr pr2))))))

(define (partB s1 s2)
  (lambda ()
    (let ([pr1 (s1)]
          [pr2 (s2)])
      (if (car pr1)
          (cons (car pr1) (partB (cdr pr1) (cdr pr2)))
          (cons (car pr2) (partB (cdr pr1) (cdr pr2)))))))
```

(c) This can also be done with a letrec inside the thunk.

```
(define (partC s)
  (lambda ()
     (let ([pr (s)])
        (if (car pr)
            (cons (car pr) (partC (cdr pr)))
            ((partC (cdr pr)))))))
```

Name:____

3. (**15** points) For each of the Racket expressions below, indicate what, if anything, the expression prints (*not* what the result is) when the expression is run in the scope of these definitions:

```
(define (a-fun x)
  (let ([y x])
    (+ y x)))

(define-syntax a-macro
  (syntax-rules ()
    [(a-macro x)
     (let ([y x])
       (+ y x))]))

(define y 17)
(define z 42)
```

(a) `(a-fun (begin (print 17) 42))`

(b) `(a-macro (begin (print 17) 42))`

(c) `(a-fun (begin (print y) z))`

(d) `(a-macro (begin (print y) z))`

(e) `(lambda() (a-fun (begin (print y) z)))`

(f) `(lambda() (a-macro (begin (print y) z)))`

Solution:

(a) 17

(b) 1717

(c) 17

(d) 1717 (this is the tricky one, Racket's macros are hygienic)

(e) prints nothing

(f) prints nothing

Name:____

4. (**15** points) Suppose you are grading a student's interpreter for the MUPL assignment and you suspect that the student made the classic error of evaluting a closure's function body in the environment where the closure is used instead of where the closure is defined. Give a MUPL test program that will work as follows: If the student got closures right, then passing your answer to `eval-prog` will evaluate to `(int 17)`, but if they made the classic error, it will raise an undefined-variable error.

   Assume other cases of the interpreter (particularly the cases for let-expressions and variables) are correct. Here are some of the struct definitions for the MUPL language in Racket; these should be plenty to answer the problem.

   Remember: The answer to the question is a MUPL program.

```
(struct var  (string) #:transparent)  ;; a variable, e.g., (var "foo")
(struct int  (num)    #:transparent)  ;; a constant number, e.g., (int 17)
(struct fun  (nameopt formal body) #:transparent) ;; a recursive(?) 1-argument function
(struct call (funexp actual)       #:transparent) ;; function call
(struct mlet (var e body) #:transparent) ;; a local binding (let var = e in body)

(define (eval-prog p) ...)
```

   Solution:
   Of course there are may solutions, but the natural approach is to call a function that has a free variable that is no longer in scope. Here is a particularly short example:

```
  (call (mlet "x" (int 17) (fun #f "y" (var "x"))) (int 0))
```

   Examples with currying can also work well provided the inner function uses the outer function's parameter, e.g.,:

```
  (call (call (fun #f "x" (fun #f "y" (var "x"))) (int 17)) (int 0))
```

Name:____

5. For this problem, consider the purpose of the Java type system to be ensuring that no "field missing" or "method missing" errors occur at run-time. Consider a change to Java where we allow methods to be called with too many arguments, e.g., 4 arguments to a 2-argument method. The typing rule is that any "extra" arguments must have some type, but any type is okay. The evaluation rule is that the extra arguments are evaluated and the results ignored.

   (a) (**5** points) Does this modified version of Java have a *sound* type system? Explain your answer, and include the definition of soundness.

   (b) (**5** points) Does this modified version of Java have a *complete* type system? Explain your answer, and include the definition of completeness.

   (c) (**5** points) Give one objective reason in favor and one objective reason against making this modification to Java.

Solution:

(a) Yes, the type system is (still) sound. A sound type system never accepts a program that, when run, may do what the type system aims to prevent. In this problem, we allow more programs to type-check, but they cannot lead to an error since the extra arguments type-check (so evaluating them won't lead to an error) and we only allow methods to be called with the same arguments as before.

(b) No, the type system is (still) incomplete. A complete type system never rejects a program that, when run, will not do what the type system aims to prevent. Even after this modification, there are many sources of incompleteness in Java's type system. For example, it would reject the program where the body of `main` is `if(false) m();` if there is no method `m`, but this program runs correctly (it does nothing).

(c) A few arguments in favor:

- This allows more programs without breaking soundness. It could even be useful. For example, we could evolve a method to take fewer arguments without breaking any existing callers.
- It lets you more concisely perform some side-effect after evaluating the arguments a function needs but before starting the function call.
- It lets you use the same list of arguments for multiple calls even if those calls take a different number of arguments.

A few arguments against:

- Method calls with too many arguments are likely errors, so it would be good to catch these errors at compile-time.
- It complicates resolving calls when multiple methods can have the same name, as with static overloading.
- It encourages (or at least allows) an unclear programming style where arguments are being evaluated but their results implicitly ignored.

Name:____

6. (**12** points) This problem considers this Ruby class:

```
class A
  attr_accessor :x
  def m1
    @x = 4
  end
  def m2
    m1
    @x > 4
  end
  def m3
    @x = 4
    @x > 4
  end
  def m4
    self.x = 4
    @x > 4
  end
end
```

(a) Is it possible to define a class `B` such that evaluating `B.new.m2` causes the method `m2` defined in class `A` (not an override of `m2`) to return `true`? If so, define class `B` as such, else explain why it is not possible.

(b) Is it possible to define a class `B` such that evaluating `B.new.m3` causes the method `m3` defined in class `A` (not an override of `m3`) to return `true`? If so, define class `B` as such, else explain why it is not possible.

(c) Is it possible to define a class `B` such that evaluating `B.new.m4` causes the method `m4` defined in class `A` (not an override of `m4`) to return `true`? If so, define class `B` as such, else explain why it is not possible.

Solution:

(a) Yes, it is possible, e.g.,

```
class B < A
  def m1
    @x = 7
  end
end
```

(b) No, it is not possible. The line `@x = 4` assigns to the instance variable that is read on the next line. There are no intervening method calls to change the value of `@x` (except for the `>` call,but that call is on `4`, which cannot be changed by the definition of class `B` — well, there probably is some way in Ruby and answers along these lines can have full credit).

(c) Yes, it is possible, e.g.,

```
class B < A
  def x= a
    @x = 7
  end
end
```

Name:____

7. (**12** points) Ruby collection classes that include the `Enumerable` mixin get many methods that are implemented in `Enumerable` using only the `each` method of `self`, which recall takes a one-argument block. One of the methods in `Enumerable` is `max`, which returns the maximum element of the collection assuming that elements of the collection can be compared with `>`. Show one way that the `Enumerable` mixin could define `max`. In your implementation, raise an error if `max` is used on an empty collection.

   The hard part of the problem is using no methods other than `each` and `>` and handling the first element correctly. The sample solution is over 15 lines, but all the lines are very short.

   Solution:

```
module Enumerable
  def max
    first = true
    sofar = nil
    each {|x|
      if first
        first = false
        sofar = x
      elsif x > sofar
        sofar = x
      end
    }
    if first
      raise "empty collection"
    else
      sofar
    end
  end
end
```

Name:____

8. (**15** points) In this problem, suppose we add record subtyping and function subtyping to ML. Because ML records are immutable (there is no way to assign to a field after a record is created), depth subtyping is sound for records. So assume record subtyping supports width, permutation, and depth, and that function subtyping supports contravariant agruments and covariant results.

   For each of the following function calls, decide if the call should type-check, answering "Yes" if it should type-check and "No" if it should not. If your answer is, "No," give a possible implementation of the relevant functions so that the call would read a field of a record that does not exist.

   In your solutions, you may use `e.f` to read field `f` rather than ML's `#f e` syntax.

```
(* assume these variables are bound to functions with the given types; they are used below *)
val f1 : { a:int, b : { c:int, d:int } } -> { a:int } = ...
val f2 : { a:int } -> { a:int, b : { c:int, d:int } } = ...
val f3 : { a:int, b : { c:int, d:int } } -> { a:int, b : { c:int, d:int } } = ...
val f4 : (({ a:int, b : { c:int} } -> { a:int }) * int) -> { a:int } = ...

val r1 : { a:int } = { a = 1 }
val r2 : { a:int, b : { c:int} } = { a=1, b = { c=2 } }
val r3 : { a:int, b : { c:int, d:int}, e:int } = { a=1, b = { c=2, d=3}, e=4 }
val r4 : { a:int, b : { c:int, d:int, e:int }} = { a=1, b = { c=2, d=3, e=4} }
```

   (a) `f1 r1`
   (b) `f1 r2`
   (c) `f1 r3`
   (d) `f1 r4`
   (e) `f2 r1`
   (f) `f2 r2`
   (g) `f2 r3`
   (h) `f2 r4`
   (i) `f4(f1,42)`
   (j) `f4(f2,42)`
   (k) `f4(f3,42)`

   Solution:

   (a) No, e.g., `val f1 = fn x => { a=x.b }`
   (b) No, `val f1 = fn x => { a=x.b.d }`
   (c) Yes
   (d) Yes
   (e) Yes
   (f) Yes
   (g) Yes
   (h) Yes
   (i) No, e.g., `val f1 = fn x => { a=x.b.d }` and `val f4 = fn(f,x) => f {a=1,b={c=2}}`
   (j) Yes
   (k) No, e.g., `val f3 = fn x => { a = x.b.d, b = {c=1,d=1} }` and
   `val f4 = fn(f,x) => f {a=1,b={c=2}}`