Name:_____

# CSE341, Spring 2013, Final Examination
# June 13, 2013

## Please do not turn the page until 8:30.

Rules:

- The exam is closed-book, closed-note, except for **both sides** 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 **unevenly** among **8** questions (many with multiple parts).
- When writing code, style matters, but do not 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. (**16** points)

   (a) Without using any helper functions, write a Racket function `filter-increasing`, which works as follows:

   - It takes three arguments (recall Racket supports multi-argument functions directly): (1) a function `f` that takes list elements and, we assume, returns numbers, (2) a number `i`, and (3) a list `xs`.
   - It returns a list that contains a subset of the elements in `xs` in the same order they appear in `xs`.
   - An element of `xs` is in the output if and only if `f` applied to the element produces a number greater than `i` and greater than the number produced by `f` for all elements earlier (closer to the head) in the list.

   (b) Write a Racket function `filter-function-maker` that takes two arguments that are the same as `f` and `i` in part (a) and returns a *function* that takes a list `xs` and returns the result of `filter-increasing` with `f`, `i`, and `xs`. Use your answer to part (a) as appropriate.

   (c) Write a call to `filter-function-maker` that returns a function that works as follows:

   - It filters out any list element that is not a positive number.
   - It includes any positive numbers that are greater than previous positive numbers in the list.
   - For example, the result for `(list 0 3 2 5 (list 19 24) #f 7 3)` would be `'(3 5 7)`.

Name:_____

2. (**11** points) For each of the following programs, indicate what would be printed. Notice that in each program there are two print expressions and there are two calls to the function.

(a)
```
(define (f1 x)
      (begin
       (print x)
       (set! x (+ x 7))
       (print x)))

    (f1 12)
    (f1 12)
```

(b)
```
(define f2
      (let ([x 5])
        (lambda (a)
           (begin
            (print x)
            (set! x (+ x 7))
            (print x)))))
    (f2 12)
    (f2 12)
```

(c)
```
(define f3
      (let ([x 5])
         (begin
            (print x)
            (lambda (a)
              (begin
               (set! x (+ x 7))
               (print x))))))
   (f3 12)
   (f3 12)
```

Name:_____

3. (**13** points) Write a Racket function `some-multiple-of-n` that works as follows:

   - It takes a stream `s` and a number `n`. (Remember a stream is a thunk that returns a pair where the cdr is a stream.)
   - It returns a list. The length of the list will be $k$ times `n` for some $k \geq 1$.
   - The returned list contains elements from the stream in order. It always contains the first `n` elements. If the $\texttt{n}^{th}$ element *is* `#f`, then the list contains the `#f` and another `n` elements. This continues until the first $k$ for which the $k \cdot \texttt{n}^{th}$ stream element is *not* `#f`, in which case this stream element is in the returned list as the last element.

   Hint: You will need a recursive helper function that takes either two or three arguments, depending on how you organize your code.

Name:_____

4. (**14** points) Here is the skeleton of the MUPL language and interpreter, including all the struct definitions that define the syntax of the language:

```
(struct var  (string) #:transparent)  ;; a variable, e.g., (var "foo")
(struct int  (num)    #:transparent)  ;; a constant number, e.g., (int 17)
(struct add  (e1 e2)  #:transparent)  ;; add two expressions
(struct ifgreater (e1 e2 e3 e4)    #:transparent) ;; if e1 > e2 then e3 else e4
(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)
(struct apair (e1 e2)     #:transparent) ;; make a new pair
(struct fst  (e)    #:transparent) ;; get first part of a pair
(struct snd  (e)    #:transparent) ;; get second part of a pair
(struct aunit ()    #:transparent) ;; unit value -- good for ending a list
(struct isaunit (e) #:transparent) ;; evaluate to 1 if e is unit else 0
(struct closure (env fun) #:transparent)

(define (envlookup env str) ...)
(define (eval-under-env e env) ...)
(define (eval-exp e)
  (eval-under-env e null))
```

Below are four wrong calls to the MUPL interpreter. For each, identify which of these choices best describes what goes wrong and **explain your answer** in 1-3 English sentences by explaining what happens when the program runs. (There is room on the next page for your answers.)

A. Racket will raise an error before `p` is defined, so `eval-exp` will never get called.

B. `eval-exp` will get called, but with something that is not a legal MUPL program.

C. `eval-exp` will get called with a legal MUPL program, but evaluation will encounter a MUPL dynamic type-error.

(a)
```
(define p (isaunit (ifgreater (int 1) (aunit) (int 3) (aunit))))
(eval-exp p)
```

(b)
```
(define p (isaunit? (ifgreater (int 1) (aunit) (int 3) (aunit))))
(eval-exp p)
```

(c)
```
(define p (aunit (ifgreater (int 1) (aunit) (int 3) (aunit))))
(eval-exp p)
```

(d)
```
(define p (aunit? (ifgreater (int 1) (aunit) (int 3) (aunit))))
(eval-exp p)
```

Name:_____

*Extra room for answers to Problem 4.*

Name:_____

5. (**12** points) In this problem, we assume the purpose of the Java type system is to prevent "method missing" errors at run-time. We consider what would happen if we *removed interfaces from the language*, i.e., the language still had classes but there is simply no notion of interfaces or classes implementing them. For each answer below, explain your answer in 1–3 English sentences.

   (a) Would this revised language have a sound type system?

   (b) Would this revised language have a complete type system?

   (c) Are interfaces enough like Ruby mixins that implementing something like Ruby's `Comparable` or `Enumerable` in Java is easier with interfaces in the language than without?

   (d) If we added multiple inheritance at the same time we removed interfaces, how could we encode the same concept as interfaces?

Name:_____

6. (**6** points) Write a Ruby method called `curry` that takes one argument and works as follows:

   - We assume the argument is an instance of `Proc` with a `call` method that expects two arguments.
   - It returns an instance of `Proc` that is a curried version of the argument: it takes one argument and returns another `Proc`.
   - For example, this use of `curry` should assign `16` to `v3`:

```
v1 = curry (lambda {|a,b| a + b})
v2 = v1.call 7
v3 = v2.call 9
```

Name:_____

7. (**15** points) Consider the following silly ML code:

```
datatype the_type = A of string | B of int | C of int * int

fun f x =
    case x of
       A s => s ^ " is coming!"
     | _ => ""

fun g (x,y) =
    case x of
       B i => i + y
     | C(i,j) => i + j + y
     | A _ => y

val foo = (f(A "summer"), g(A "winter",7))
```

(a) What is `foo` bound to after this program is evaluated?

(b) Port this code to Ruby as follows:

- Use an OOP style with multiple class definitions. Do not define methods outside of these classes.
- Use `initialize` methods, other methods, and instance variables as appropriate.
- Have `foo` hold a two-element array.

(There is more room on the next page in case you need it.)

Name:_____

*Extra room for answers to Problem 7.*

Name:_____

8. (**13** points) In this problem, we consider a language like in lecture containing (1) records with mutable fields, (2) higher-order functions, and (3) subtyping. We do *not* require explanations for your answers.

   (a) For each of the following questions, answer "yes" if and only if the proposed subtyping relationship is sound, meaning it would not allow a program to type-check that could then try to access a field in a record that did not have that field.

      i. Is `{f1 : string, f2 : string}` a subtype of `{f1 : string, f2 : string, f3 : string}` ?

      ii. Is `{f1 : string, f2: {g1 : string, g2 : string} }` a subtype of `{f1 : string, f2 : {g1 : string} }` ?

      iii. Is `string -> {f1 : string, f2 : int, f3 : int}` a subtype of `string -> {f3 : int, f2 : int}` ?

      iv. Is `{f1 : string, f2 : int, f3 : int} -> string` a subtype of `{f3 : int, f2 : int} -> string` ?

      v. Is `int -> {f1 : string, f2 : {g1 : string} }` a subtype of `int -> {f1 : string, f2 : {g1 : string, g2 : string} }` ?

   (b) If we change the language so that records are immutable (you cannot update contents of a field), which, if any, of your answers to part (a) change?