Name:_____

CSE341 Spring 2016, Final Examination
June 6, 2016

Please do not turn the page until 8:30.

Rules:

- The exam is closed-book, closed-note, etc. except for *both* sides of one 8.5x11in piece of paper.
- **Please stop promptly at 10:20.**
- There are **125 points**, distributed **unevenly** among **9** questions (all with multiple parts):
- **The exam is printed double-sided.**

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. (**16** points) (Racket programming) In this problem, the lists we consider have as elements only immutable pairs (built with `cons`) of numbers or mutable pairs (built with `mcons`) of numbers. Here are examples of expressions that create such lists:

```
(list (cons 12 27) (mcons 13 9) (cons 10 15))
(list (cons 1 20) (cons 4 19))
null
```

   (a) Define a Racket function `sum-all-pairs` that takes a list and returns the sum of all the numbers contained anywhere in it. (The examples above have sums 86, 44, and 0.)

   (b) Define a Racket function `negate-lefts` that works as follows (pay attention, it is a bit strange):

   - It takes a list and returns a new list of the same length.
   - If the i^{th} element of the argument is `(cons` x y`)`, then the i^{th} element of the result is a *new* cons cell holding the negation of x and (not-negated, i.e., unchanged) y.
   - If the i^{th} element of the argument is `(mcons` x y`)`, then *mutate* the pair so that it holds the negation of x in its mcar and use the *same* pair in the result.

   (c) Define a Racket variable `part-c` such that you would get this behavior in DrRacket's REPL:

```
> part-c
(list (mcons 1 2) '(3 . 4) (mcons 1 2))
> (negate-lefts part-c)
(list (mcons -1 2) '(-3 . 4) (mcons -1 2))
```

   (d) Define a Racket variable `part-d` such that you would get this behavior in DrRacket's REPL:

```
> part-d
(list (mcons 1 2) '(3 . 4) (mcons 1 2))
> (negate-lefts part-d)
(list (mcons 1 2) '(-3 . 4) (mcons 1 2))
```

Name:_____

2. (**10** points) Consider this (silly) Racket code:

```
(define y 7)

(define (f x)
  (let ([y 2])
    (+ y x)))

(define g
  (let* ([x y]
         [y (+ x 2)]
         [f (lambda (x) (+ y x))])
    f))

(define a (f y))

(define b (g y))

(set! y 0)

(define c (f y))

(define d (g y))
```

After all this code is evaluated:

(a) What is `a` bound to?

(b) What is `b` bound to?

(c) What is `c` bound to?

(d) What is `d` bound to?

Name:_____

3. (**14** points) (Streams) Remember a stream is a thunk that returns a pair where the cdr is a stream.

   (a) Write a Racket function `list->sticky-stream` that works as follows:

   - It takes a list, which you can assume is non-empty.
   - If the list has length n, then:
     - For $i < n$, the i^{th} element of the stream is the i^{th} element of the list (counting from 1).
     - For $i \geq n$ the i^{th} element of the stream is the last element of the list.

   Do not use any arithmetic in your solution.

   (b) Write a Racket function `stream-until-repeat` that works as follows:

   - It takes a stream and returns a list.
   - The list is some *prefix* of the stream, i.e., if the list has length n, then it contains the first n values produced by the stream in order.
   - The list is as long as it can be without any two *adjacent* elements being equal (as defined by Racket's `equal?`). That is, the list can have duplicated elements, but not next to each other.

Name:_____

4. (**10** points) Each of the Racket expressions below causes an error when evaluated. Give a brief description of what the error is. We won't grade on how similar your wording is to DrRacket's REPL provided you communicate the right idea. For example, if an expression is `(cdr null)`, you could answer, "`cdr` requires a pair, not an empty list."

   (a) `((+ 3 4))`

   (b) `(let ([x 2]) (let ([x 3][x (+ x 4)]) x))`

   (c)
   ```
   (define-syntax hd
     (syntax-rules ()
       [(hd x) ((car) (x))]))

   (hd (lambda () (cons 7 7)))
   ```

   (d) `(error "this is my special final-exam error")`

   (e) `((error "this is my special final-exam error"))`

Name:_____

5. (**17** points) For your reference, here are the Racket structs we used to define MUPL's abstract syntax and the start of `eval-under-env`.

```
(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 isgreater (e1 e2)    #:transparent) ;; if e1 > e2 then 1 else 0
(struct ifnz (e1 e2 e3) #:transparent) ;; if not zero e1 then e2 else e3
(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 first   (e)     #:transparent) ;; get first part of a pair
(struct second  (e)     #:transparent) ;; get second part of a pair
(struct munit   ()      #:transparent) ;; unit value -- good for ending a list
(struct ismunit (e)     #:transparent) ;; if e1 is unit then 1 else 0
```

```
(define (eval-under-env e env)
  (cond ... ;; one case for each kind of MUPL expression
        ))
```

(a) We want to add a swap expression to MUPL:

```
(struct swap (e) #:transparent)
```

A swap expression evaluates its subexpression. If the result is a MUPL pair value, then swap produces a new pair where the first componenet is the subresult's second component and the second component is the subresult's first component.

Write the case you would add to `eval-under-env` to implement MUPL swap expressions.

(b) Now suppose we do not add the `swap` struct or change `eval-under-env`. Make it "feel like" MUPL has swap expressions by defining a Racket function `swap` that is a MUPL macro. To avoid unneeded recomputation, assume the expression passed to `swap` does not use MUPL variable `_x`.

(c) Here is an incorrect attempt to write a length function in MUPL for MUPL lists:

```
(define mupl-length-wrong
  (fun "f" "x" (ifnz (munit? (var "x"))
                     (int 0)
                     (add (int 1) (call (var "f") (second (var "x")))))))
```

i. What is wrong with the code above? Describe how to fix the MUPL code so that the Racket variable `mupl-length-wrong` is bound to a correct MUPL length function.

ii. What would happen when trying to evaluate:

```
(eval-exp (call mupl-length-wrong (apair (int 86) (munit))))
```

Briefly explain your answer.

Name:_____

More room, if needed, for your Problem 5 answer.

Name:_____

6. (**13** points) In ML, the `hd` and `tl` functions can be used to access list parts (as we did in Homework 1).

   (a) What are the types of `hd` and `tl`?

   (b) Does ML's type system prevent applying `hd` and `tl` to a number?

   (c) Does ML's type system prevent applying `hd` and `tl` to the empty list?

   (d) If we changed ML's type system so that `hd` and `tl` could be used only on lists of numbers:

      i. Would your answer to part (b) change?

      ii. Would your answer to part (c) change?

   (e) Consider these alternate definitions:

```
fun hd_alt xs =
   case xs of
       [] => []
     | x::y => x
```

```
fun tl_alt xs =
   case xs of
       [] => []
     | x::y => y
```

      i. What type, if any, would ML give to `hd_alt`?

      ii. What type, if any, would ML give to `tl_alt`?

      iii. Would it be sound (in terms of what ML's type system is designed to prevent) to give `hd_alt` the type for `hd` you gave in part (a)?

      iv. Would it be sound (in terms of what ML's type system is designed to prevent) to give `tl_alt` the type for `tl` you gave in part (a)?

Name:_____

7. (**14** points) Recall Ruby's `Enumerable` module is a mixin that adds lots of useful functionality to a class by relying on the class' `each` method. Further recall `each` takes no regular arguments and a block that takes one argument.

   (a) Add to the `Enumerable` module a method `increasing?` that takes one regular argument `i` and does not expect a block. `increasing?` should return `true` if the first value produced by `each` is (strictly) greater than `i` and each subsequent value produced by `each` is (strictly) greater than the previous one. Else `increasing?` should return `false`. Here are examples (since `Array` includes `Enumerable`):

   - `[3,7,9].increasing? 2` evaluates to `true`
   - `[3,7,9].increasing? 4` evaluates to `false`
   - `[3,9,7].increasing? 2` evaluates to `false`

   (b) Will your part (a) solution cause an error with `[x].increasing? 2` if `x` is bound to an object that is not a number? Answer "always", "sometimes", or "never" and *briefly* explain your answer.

   (c) Add to the `Enumerable` module a method `count_same` that takes no regular arguments and expects a block that expects one argument. `count_same` should return the *number* of elements produced by `each` for which the element is equal to (using `==`) the result of passing the element to the block passed to `count_same`. Here are examples:

   - `[1,3,0,4].count_same {|x| x * x}` evaluates to `2`.
   - `[1,3,0,4].count_same {|x| x + 1}` evaluates to `0`.
   - `[1,3,0,4].count_same {|x| 3}` evaluates to `1`.

Name:_____

8. (**14** points) This code defines two Ruby classes, including `same_size` methods that work on any of the 4 combinations of a `MyIntList` and a `MyRange`.

```
class MyRange
  def initialize(lo,hi)
    @lo = lo
    @hi = hi
  end
  # ... other methods not shown
  def min
    @lo
  end
  def max
    @hi
  end
  def same_size other
    other.same_size_range self
  end
  def same_size_range other
    other.max - other.min == max - min
  end
  def same_size_list other
    other.length == max - min + 1
  end
end
```

```
class MyIntList
  def initialize (i,r)
    @head = i
    @rest = r
  end
  # ... other methods not shown
  def length
    if @rest.nil? then 1 else 1 + @rest.length end
  end
  def same_size other
    other.same_size_list self
  end
  def same_size_range other
    other.max - other.min + 1 == self.length
  end
  def same_size_list other
    other.length == self.length
  end
end
```

(a) Complete the ML code below by writing a function `same_size : intPile * intPile -> bool` to port the Ruby code to ML in a functional style. Use the functions defined below, but do not define any additional helper functions.

```
datatype myIntList = Nil
                   | List of int * myIntList
type myRange = int * int
datatype intPile = L of myIntList
                 | R of myRange
```

```
fun length xs =
  case xs of
      Nil => 0
    | List(_,xs) => 1 + length xs
fun max (_,hi) = hi
fun min (lo,_) = lo
```

(b) The style of the Ruby code is unnecessarily complicated even if we want a "pure OOP" approach. Give alternate definitions of `same_size` as follows:

- Do *not* use `is_a?`, `instance_of?`, `class`, or similar methods.
- Do *not* use `same_size_range` or `same_size_list`.
- *Do* define a helper method in each class (unless you decide it is unneeded for a class).
- *Do* redefine `same_size` in each class.

Name:_____

More room, if needed, for your Problem 8 answer.

Name:_____

9. (**17** points) This problem considers a language like in lecture containing (1) records with mutable fields, (2) higher-order functions, and (3) subtyping. Like in lecture, subtyping for records includes width subtyping and permutation subtyping but not depth subtyping, and subtyping for functions includes contravariant arguments and covariant results. The goal of the type system is to prevent accessing a field of a record that does not exist.

   We have these functions defined:

```
fun f1 (x : {foo : int, bar : int}) =
    x.foo = 7;
    x.bar = x.bar + 1
fun f2 (x : {foo : int, bar : int}) =
    {baz = x.bar + x.foo}
fun f3 (x : {foo : {foo : int}, bar : int}) =
    x.foo.foo - 42
fun f4 (g : {foo : int, bar : int} -> {foo : int}) =
    g {foo = 17, bar = 19}
```

   (a) Answer one of (A)-(D) below for this expression:

   (part i.) `f1 {foo = 19, bar = 20, baz = 24}`

   (A) It type-checks and, if run, does not access a record field that does not exist.
   (B) It does not type-check and, if run, does not access a record field that does not exist.
   (C) It type-checks and, if run, accesses a record field that does not exist.
   (D) It does not type-check and, if run, accesses a record field that does not exist.

   Repeat the exercise (i.e., answer one of (A)-(D)) for each of these expressions:

   ii. `f1 (f2 {foo = 19, bar = 20})`
   iii. `f2 (f1 {foo = 19, bar = 20})`
   iv. `f3 {foo = { foo = 24, bar = 25 }, bar = 26}`
   v. `f4 (fn y => {foo = y.foo, bar = y.bar + y.baz})`
   vi. `f4 (fn y => {foo = y.foo - 2})`
   vii. `f4 (fn y => {}) (* {} is a record with zero fields *)`
   viii. `f4 (fn y => {foo = y.foo, bar = y.foo})`

   (b) Complete the following blanks with "is definitely", "might or might not be", or "is definitely not" where (A)-(D) refer to the statements in part (a).

   i. If no expression in a language has answer (B), the type system _____ sound.
   ii. If no expression in a language has answer (C), the type system _____ sound.
   iii. If no expression in a language has answer (D), the type system _____ sound.

Name:_____

Here are two extra (sides of) pages in case you need them. If you use them for a question, please write "see also extra sheets" or similar on the page with the question.

Name:_____

Second extra page.