Name:_____

CSE341 Autumn 2018, Final Examination
December 13, 2018

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 **8** 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. (**21** points) (Racket programming)

Please put your answers on the next page.

(a) Write a Racket function `map-index1` that behaves as follows:

- Like `map` it takes two arguments, a function and a list, and produces a list of the same length where the function is applied to each list element in order.
- Unlike `map`, the function passed to `map-index1` takes two arguments, first a number and second a list element.
- When the function passed to `map-index1` is called, the first argument is i when the second argument is the i^{th} element of the list.
- The first list element is at position 1.

Use one locally-defined helper function (using `letrec` or a local define) and no other helper functions.

(b) Use `map-index1` to write a Racket function `redact-evens` that takes a list of strings and returns a list of strings. The strings at odd-numbered list positions in the input list are in the output list unchanged and the strings at even-numbered list positions in the input list are replaced in the output list by the empty string `""`. Hints:

- `map-index1` is not curried, so you will not be able to use partial application.
- The mod operation (e.g., `%` in Java) is `remainder` in Racket.
- `(redact-evens (list "hi" "bye" "foo" "bar" "quux"))` should evaluate to `'("hi" "" "foo" "" "quux")`.

(c) *Without* using `map-index1`, write a Racket function `add-index` that takes a list and returns a list of the same length. The output list is a list of pairs (cons cells) where each pair's car is the position of the list (starting at 1) and the cdr is the element at the same position of the input list. For example, `(add-index (list "hi" "bye" "foo" "bar" "quux"))` should evaluate to `'((1 . "hi") (2 . "bye") (3 . "foo") (4 . "bar") (5 . "quux"))`.
Use one locally-defined helper function (using `letrec` or a local define) and no other helper functions.

(d) Fill in the blanks so that `map-index2` is a suitable replacement for `map-index1`. Note `map` is in Racket's standard library.

```
(define (map-index2 f xs)
  (map (lambda (x) ______________________________) (add-index ________________)))
```

(e) Is your implementation of `map-index2` equivalent (in the sense we discussed in class) to your implementation of `map-index1` for *all* `f` and `xs`, *some* `f` and `xs`, or *no* `f` and `xs`? ***Explain your answer*** in roughly 1 English sentence.

Name:_____

Please put your answers to Problem 1 here.

Name:_____

2. (**14** points) (Scope and Mutation)

(a) Consider the following Racket definitions:

```
(define x 15)
(define y 17)
```

In an environment where `x` and `y` are defined as above, for each of the following bindings, either give the value that the variable would be bound to after evaluation or indicate "error" if an error would occur. For example, the answer for `(define b0 (+ x y))` would be 32.

i.
```
(define b1 (let ([x y]
                 [y x])
             (cons x y)))
```

ii.
```
(define b2 (letrec ([x y]
                    [y x])
             (cons x y)))
```

iii.
```
(define b3 (let* ([x y]
                  [y x])
             (cons x y)))
```

iv.
```
(define b4 (let ([x (cons x y)]
                 [y (cons x y)])
             (cons (car x) (car y))))
```

(b) Consider the following Racket program:

```
(define f
  (let ([x 1])
    (begin (set! x (+ x 1))
           (lambda ()
             (let ([y 1])
               (begin (set! y (+ y 1))
                      (set! x (+ y x))
                      (lambda ()
                        (let ([z 1])
                          (begin (set! z (+ z 1))
                                 (set! y (+ y z))
                                 (set! x (+ y x))
                                 x)))))))))
(define a1 (f))
(define a2 (f))
(define p1 (a1))
(define p2 (a2))
(define p3 (a1))
(define p4 (a2))
```

i. After executing this program, what is `p1` bound to?

ii. After executing this program, what is `p2` bound to?

iii. After executing this program, what is `p3` bound to?

iv. After executing this program, what is `p4` bound to?

Name:_____

3. (**20** points) (Streams) Recall we defined a stream as a thunk that returns a pair where the cdr is a stream. We now call such a stream a *regular stream*, so that we can define an *endable stream* to be a thunk that can either return a pair where the cdr is a stream or return `#f` instead of a pair. Returning `#f` means the endable stream has no more elements. So an endable stream may or may not have an infinite number of elements.

   (a) Write a Racket function `sum-until-non-number-or-end` that takes an endable stream and either returns a number or runs forever. The number is the sum of all numbers appearing before the first non-number in the stream or before the stream's end (if the stream ends before a non-number). Hint: `number?`

   (b) Consider this Racket code:

```
(define (stream->endable-stream stop? s)
  (lambda ()
    (let ([p (s)])
      (if (stop? (car p))
          #f
          (cons (car p) (stream->endable-stream stop? (cdr p)))))))

(define (s-help i) (cons i (lambda () (s-help (* 2 i)))))
(define s (lambda () (s-help 1)))
```

   For each of the following, indicate what the expression evaluates to, or "does not terminate" if it does not terminate or "error" if it ends due to an error.

   i. `(sum-until-non-number-or-end (stream->endable-stream (lambda (y) (> y 10)) s))`

   ii. `(sum-until-non-number-or-end (stream->endable-stream (lambda (y) (> y 0)) s))`

   iii. `(sum-until-non-number-or-end (stream->endable-stream (lambda (y) #f) s))`

   (c) Write a Racket function `zip-endable-streams` that takes two endable streams `s1` and `s2` and returns an endable stream. The returned endable stream contains pairs where the i^{th} pair produced by the stream has the i^{th} element of `s1` in the car and i^{th} element of `s2` in the cdr. The returned stream is infinite if both argument streams are infinite, else it ends when the shorter of the two argument streams ends.

Name:_____

4. (**17** points) (Interpreter implementation) Below is some of the code we provided you for Homework 5 (MUPL). See the next page for the questions.

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

(define (eval-under-env e env)
  (cond [(var? e)
         (envlookup env (var-string e))]
        [(int? e)
         e]
        [(add? e)
         (let ([v1 (eval-under-env (add-e1 e) env)]
               [v2 (eval-under-env (add-e2 e) env)])
           (if (and (int? v1)
                    (int? v2))
               (int (+ (int-num v1)
                       (int-num v2)))
               (error "MUPL addition applied to non-number")))]
        [(isgreater? e)
         (let ([v1 (eval-under-env (isgreater-e1 e) env)]
               [v2 (eval-under-env (isgreater-e2 e) env)])
           (if (and (int? v1)
                    (int? v2))
               (if (> (int-num v1) (int-num v2))
                   (int 1)
                   (int 0))
               (error "MUPL isgreater applied to non-number")))]
   ...))
```

Name:_____

(a) We can extend the MUPL language with an expression form for computing the max of two subexpressions with this struct:

```
(struct mmax (e1 e2) #:transparent)
```

But we give a semantics more flexible than some other features in MUPL:

- If both subexpressions evaluate to MUPL ints, then return their max.
- If only one subexpression evaluates to a MUPL int, return that MUPL int.
- Raise a dynamic error only if both subexpressions do not evaluate to MUPL ints.

Implement this by adding a case to the "big `cond`" in `eval-under-env`.

(b) Alternately, we could use a Racket function like a MUPL macro for providing mmax, but *without* the more flexible semantics from part (a). Implement a Racket function `mmax2` such that `(mmax2 e1 e2)` produces a MUPL expression that, when run, evaluates to the maximum of the results of the subexpression, but encounters a dynamic error unless both subexpressions produce numbers. Use `mlet` so that the expression produced evaluates `e1` and `e2` only once; to do so, assume you can use variables `"_x"` and `"_y"` without shadowing problems.

(c) Explain in roughly 1 English sentence why the limited features available in MUPL make it so the macro approach in part (b) cannot provide the more flexible semantics from part (a). What feature would MUPL need to make it possible?

Name:_____

5. (**16** points) (Static Typing) In this problem, we consider ML's type system and assume the purpose of the type system is to prevent passing the wrong kind of value to a primitive, such as trying to multiply a function. (In practice, the type system is intended to prevent more, but that is not really relevant here.)

   (a) In 1-2 English sentences, what is ML's typing rule for expressions of the form `if e1 then e2 else e3`?

   (b) Is ML's type system sound (no explanation required)?

   (c) Is ML's type system complete (no explanation required)?

   (d) Fill in the blank such that this ML binding does not type-check:

```
val x = 13 + (if true then 4 else _____________)
```

   (e) In 1-2 English sentences, propose a change to the type system that:
   - Allows all the programs that used to type-check to still type-check
   - Does not let a program to type-check that did not type-check before if that program could pass the wrong kind of value to a primitive.
   - Does allow a set of programs to type-check that did not type-check before, *including your answer to part (d).*

   (f) With your proposed change, is ML's type system sound (no explanation required)?

   (g) With your proposed change, is ML's type system complete (no explanation required)?

Name:_____

6. (**13** points) (Ruby blocks and mixins) Recall the `Enumerable` mixin provides many useful methods by assuming that any class including the mixin defines an `each` method that takes a block and iterates over "its elements" (the notion of "its elements" depends on the class), passing each element to the block.

   One method in `Enumerable` is `any?`, which takes a block and returns `true` if the block evaluates to `true` (or anything "not false") for any of "the elements".

   (a) Show how `any?` can be defined in the `Enumerable` mixin. (Unlike the real `any?` in Ruby, your solution can assume callers always provide a block. We also do not specify if `any?` stops as soon as it determines the answer is true or not.)

   (b) Add an appropriate `each` method to this class definition so that `Pair.new(a,b).any? {|x| x}` would be equivalent to `a || b`.

```
class Pair
  include Enumerable
  def initialize(x,y)
    @x = x
    @y = y
  end
  # your code here
  # but write it to
  # the right
end
```

   (c) Consider now a different mixin `E2` that is like `Enumerable` except it *provides* `each` but *assumes* that `any?` is defined (the opposite of how `Enumerable` works). Show how `each` can be defined in `E2`.

   (d) Now assume a mixin `E3` that is like `Enumerable` except it defines *both* `each` in terms of `any?` (your answer to part (c)) and `any?` in terms of `each` (your answer to part (a)). Further assume `class Pair` includes `E3` instead of `Enumerable`.

      i. Given your answer to part (b) is still in the `Pair` class, does `Pair.new(a,b).any? {|x| x}` still behave as desired? If not, what happens instead?

      ii. If you remove your answer to part (b) so that `each` and `any?` are both provided by `E3` and not overridden, does `Pair.new(a,b).any? {|x| x}` still behave as desired? If not, what happens instead?

Name:_____

7. (**12** points) (OOP) This problem considers ML code written in a functional style and Ruby written in an OOP style for the same problem. Here's the provided code for both languages:

```
datatype shirt = ShortSleeve | LongSleeve | TankTop
datatype hat = Winter | Summer | Costume
datatype upper_body_clothing = Shirt of shirt | Hat of hat | Necklace | Gloves

fun num_sleeves c =
  case c of
      Shirt s => (case s of
                    TankTop => 0
                  | _ => 2)
    | _ => 0
```

```
class UpperBodyClothing
  def num_sleeves
    0
  end
end
class Shirt < UpperBodyClothing
  def num_sleeves
    2
  end
  def good_for_hot_day
    true
  end
end
class ShortSleeve < Shirt
end
```

```
class LongSleeve < Shirt
  def good_for_hot_day
    false
  end
end
class TankTop < Shirt
  def num_sleeves
    0
  end
end
class Hat < UpperBodyClothing
  def good_for_hot_day
    false
  end
end
class Winter < Hat
end
```

```
class Summer < Hat
  def good_for_hot_day
    true
  end
end
class Costume < Hat
end
class Necklace
  def good_for_hot_day
    true
  end
end
class Gloves
  def good_for_hot_day
    false
  end
end
```

(a) The ML code for `num_sleeves` is correct but the Ruby code has a couple bugs.

i. For what objects is the Ruby code wrong?

ii. What would happen for such objects?

iii. How would you fix the bugs?

(b) The Ruby code for `good_for_hot_day` is correct. Port this code to ML by writing a function `good_for_hot_day`.

Name:_____

8. (**12** 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 : int, f2 : { a: int, b : int}, f3 : string}` a subtype of `{f2 : { a: int, b : int}, f1 : int}`?

      ii. Is `{f1 : int, f2 : { a: int, b : int}, f3 : string}` a subtype of `{f1 : int, f2 : {a : int}, f3 : string}`?

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

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

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

   (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?

Name:_____

Use this page for any answers that don't fit on another page, but please indicate on the other page that you are doing so. Write something like, "see last page."