Name:_____

CSE341 Spring 2019, Final Examination
June 13, 2019

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 **130 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. (**21** points) (Racket programming)

   (a) Write a Racket function `fold-basic` that behaves as follows:

   - It is like SML's `List.foldl` function except it does not use currying and the order of arguments is as explained below.
   - It takes three arguments: a function, an initial accumulator, and a list. (Do not check the arguments are what is expected.)
   - It repeatedly applies the function to the accumulator (first argument to the function) and the next list element (second argument to the function) to produce the next accumulator.
   - The result is the final accumulator.

   (b) Consider this function

```
(define (mystery xs)
  (fold-basic (lambda (x y) (if (or (not x) (> y x)) y x)) #f xs))
```

   For each of these calls, indicate "error" if the call fails with an error, else indicate the result of the call.

   i. `(mystery null)`

   ii. `(mystery (list 4 5 9 3 7))`

   iii. `(mystery (list 4 5 #f 3 7))`

   (c) Write a Racket function `fold-deep` that is like `fold-basic` except that if a list element is another list, it performs `fold-deep` with the same function and the current accumulator over that nested list. It should continue to "fold deeply" over any nesting of lists (lists inside lists inside lists...). For example, `(fold-deep + 0 (list 3 (list 4 (list 5) 6) (list 7)))` would evaluate to 25.

   (d) What does this expression evaluate to?

```
(fold-deep
  (lambda (x y) (cons y x))
  null
  (list (list 3 4) null (list 3 5 (list "hi") 9) (cons 4 6) (cons 5 null)))
```

Solution:

(a)
```
(define (fold-basic f acc xs)
  (if (null? xs)
      acc
      (fold-basic f (f acc (car xs)) (cdr xs))))
```

(b)
i. `#f`
ii. `9`
iii. error

(c) It is also correct for the `list?` below to be `pair?` or `cons?`.

```
(define (fold-deep f acc xs)
  (cond [(null? xs) acc]
        [(list? (car xs)) (fold-deep f (fold-deep f acc (car xs)) (cdr xs))]
        [#t (fold-deep f (f acc (car xs)) (cdr xs))]))
```

Here is an alternate solution we had not anticipated but gave full credit for:

```
(define (fold-deep f acc xs)
    (fold-basic
      (lambda (acc y)
        (if (list? y)
            (fold-deep f acc y)
            (f acc y)))
    acc xs))
```

(d) `'(5 (4 . 6) 9 "hi" 5 3 4 3)`

Name:_____

2. (**12** points) (Scope and mutation) Suppose we evaluate all the code below as a single Racket program.

```
(define x 1)
(define y 2)

(define (g a)
  (+ a y))

(define (f1 z)
  (let ([x y]
        [y x])
    (+ x (g y) z)))

(define (f2 z)
  (let* ([x y]
         [y x])
    (+ x (g y) z)))

(define (f3 z)
  (let ([t x])
    (begin
      (set! x y)
      (set! y t)
      (let ([ans (+ x (g y) z)]
            [u x])
        (begin
          (set! x y)
          (set! y u)
          ans)))))

(define part-a (f1 3))
(define part-b (f2 3))
(define part-c (f3 3))
(define part-d (f1 3))
(define part-e (f2 3))
(define part-f (f3 3))
```

(a) What is `part-a` bound to?

(b) What is `part-b` bound to?

(c) What is `part-c` bound to?

(d) What is `part-d` bound to?

(e) What is `part-e` bound to?

(f) What is `part-f` bound to?

Solution:
(a) 8 (b) 9 (c) 7 (d) 8 (e) 9 (f) 7

Name:_____

3. (**18** points) (Streams) As in class, we define a stream to be a thunk that when called returns a pair where the cdr of the pair is a stream.

   (a) Write a function `count-until-total-is-more` that behaves as follows:

   - It takes two arguments: a stream `s`, which we assume contains only non-negative numbers, and a number `limit`.
   - It returns the minimum number of stream elements that need to be summed to make the sum strictly greater than `limit`.

   For example, if `s` represents the natural numbers `1,2,3,...` and `limit` is 6, then the result should be 4. For full credit, do not define any helper functions. Hint: The base case is when `limit` is negative. **Solution:**

```
(define (count-until-total-is-more s limit)
  (if (< limit 0)
      0
      (let ([pr (s)])
        (+ 1 (count-until-total-is-more (cdr pr) (- limit (car pr)))))))
```

   (b) Write a function `every-third`, that takes a stream `s` and returns a stream that contains every third element of `s`. For example if `s` represents `1,2,3,...` then `(every-third s)` would represent `3,6,9,...`. For full credit, do not define any recursive helper functions (anonymous functions are fine). **Solution:**

```
(define (every-third s)
  (lambda ()
    (let ([third-pr ((cdr ((cdr (s)))))])
      (cons (car third-pr)
            (every-third (cdr third-pr))))))
```

   (c) Write a function `every-nth` that takes a stream `s` and a positive integer `n` and returns a stream that contains every n^{th} element of `s`. For example, `(every-nth s 3)` would be equivalent to `(every-third s)`. For full credit, use one locally defined recursive helper function.

Solution:

```
(define (every-nth s i)
  (letrec ([f (lambda (s2 j)
                (if (= j 1)
                    s2
                    (f (cdr (s2)) (- j 1))))])
    (lambda ()
      (let ([pr ((f s i))])
        (cons (car pr)
              (every-nth (cdr pr) i))))))
```

Name:_____

4. (**16** points) (Interpreter implementation) Here is some of the code we provided you for Homework 5 (MUPL).

```
(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))]
        [(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")))]
   ...))
```

Assume all the not-shown pieces are implemented correctly, including raising an appropriate error message if a subexpression evaluates to the wrong type of MUPL value.

In this question, we extend the MUPL language with this expression form:

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

It has this semantics:

- The subexpression should evaluate to a pair where the first component of the pair is a closure, else it is an error.
- The result is the result of calling the first component of the pair with the second component of the pair as the argument.

See the next page for the questions.

Name:_____

(a) Consider this implementation of the `callpair` case in "the big `cond`" of `eval-under-env`:

```
[(callpair? e) (eval-under-env (mlet "x" (callpair-e e)
                                     (call (first (var "x")) (second (var "x"))))
                               env)]
```

Answer (i) and then either (ii) or (iii) as explained below:

i. Yes or No: Is this implementation correct in cases where the subexpression evaluates to a pair whose first component is a closure?

ii. If your answer to part (i) was yes, answer this: If a callpair's subexpression evaluates to something that *is not a pair*, what will happen when the callpair is evaluated?

iii. If your answer to part (i) was no, answer this: If a callpair's subexpression evaluates to something that *is a pair*, what will happen when the callpair is evaluated?

(b) Repeat part (a) for this implemenation of `callpair`:

```
[(callpair? e) (eval-under-env (mlet "x" (callpair-e e)
                                     (call (apair-e1 (var "x")) (apair-e2 (var "x"))))
                               env)]
```

(c) Repeat part (a) for this implemenation of `callpair`:

```
[(callpair? e) (let ([pr (eval-under-env e env)])
                 (if (not (apair? pr))
                     (error "argument to callpair did not evaluate to a pair")
                     (eval-under-env (call (apair-e1 pr) (apair-e2 pr))
                                     env)))]
```

(d) Repeat part (a) for this implemenation of `callpair`:

```
[(callpair? e) (let ([pr (eval-under-env (callpair-e e) env)])
                 (if (not (apair? pr))
                     (error "argument to callpair did not evaluate to a pair")
                     (eval-under-env (call (apair-e1 pr) (apair-e2 pr))
                                     env)))]
```

Solution:

(a) (i) Yes, (ii) an error will get raised in the `first` case indicating the argument is not a pair

(b) (i) No, (iii) The call to `apair-e1` will fail because it is passed a `var`, not an `apair`.

(c) (i) No, (iii) The callpair evaluation will never terminate because evaluate `e` again rather than `callpair-e e`.

(d) (i) Yes, (ii) The error message in the code shown in part (d) will be given

Name:_____

5. (**16** points) (Soundness/Completeness) Suppose a committee is designing a new language *BestEver* and wishes to statically prevent bad thing *OhNoes* from happening in the language. Two subcommittees have presented competing type-system proposals called *LionTypes* and *TigerTypes*. The committee chair proposes two combination type systems to resolve the competition:

   - *WhyNotBoth*, a type system that accepts a program if and only if *LionTypes* and *TigerTypes* accept the program.
   - *EitherWay*, a type system that accepts a program if and only if at least one of *LionTypes* or *TigerTypes* accepts the program.

   Answer yes/no/maybe — no explanations needed.

   (a) If *LionTypes* is sound and *TigerTypes* is sound, is *WhyNotBoth* sound?

   (b) If *LionTypes* is sound and *TigerTypes* is sound, is *EitherWay* sound?

   (c) If *LionTypes* is sound and *TigerTypes* is unsound, is *WhyNotBoth* sound?

   (d) If *LionTypes* is sound and *TigerTypes* is unsound, is *EitherWay* sound?

   (e) If *LionTypes* is unsound and *TigerTypes* is unsound, is *WhyNotBoth* sound?

   (f) If *LionTypes* is unsound and *TigerTypes* is unsound, is *EitherWay* sound?

   (g) If *LionTypes* is sound and *TigerTypes* is complete, is *WhyNotBoth* complete?

   (h) If *LionTypes* is sound and *TigerTypes* is complete, is *EitherWay* complete?

   Solution:

   (a) yes
   (b) yes
   (c) yes
   (d) no
   (e) maybe
   (f) no
   (g) no (if you assume the language is Turing-complete so *LionTypes* cannot be complete) or maybe (if you do not assume that). No credit for yes.
   (h) yes

Name:_____

6. (**8** points) (Ruby programming)

Consider the code below, which adds two different methods to the `Array` class that are intended to compute whether an array is sorted, assuming its elements are comparable. (Reminder: The `shift` method returns the 0^{th} element after removing it from the array and moving all other elements "over" one position toward the beginning of the array.)

```
class Array
  def is_sorted1
    n = shift
    all? {|i| n2 = n; n = i; n2 <= i}
  end
  def is_sorted2
    a = Array.new(self)
    n = a.shift
    a.all? {|i| n2 = n; n = i; n2 <= i}
  end
end
```

(a) Does a call to `is_sorted1` always return the correct true/false result?

(b) Does a call to `is_sorted2` always return the correct true/false result?

(c) Explain in 1–2 English sentences why `is_sorted1` and `is_sorted2` are *not* equivalent.

(d) Provide two tests that show `is_sorted1` and `is_sorted2` are not equivalent. The tests should be identical except one calls `is_sorted1` wherever the other calls `is_sorted2`. Tests can just be top-level Ruby code.

Solution:

(a) Yes

(b) Yes

(c) `is_sorted2` does not mutate `self` while `is_sorted1` calls `shift` on `self` causing the element in position 0 to be removed from `self`.

(d) Many possible answers, but they need to do something observable (print something different or produce a different result)

```
a = [4,3,5]                      a = [4,3,5]
b = a.is_sorted1 # false         b = a.is_sorted2 # false
c = a.is_sorted1 # true [!!]     c = a.is_sorted2 # false
```

Name:_____

7. (**8** points) (Ruby mixins) Recall Ruby's `Comparable` mixin defines `>`, `>=`, etc.

   (a) Add a method `between` to `Comparable` that takes two arguments and evaluates to `true` if and only if the receiver is strictly greater than the first argument and strictly less than the second argument.

   (b) Add a method `less_than_all?` to `Comparable` that takes one argument and returns true if and only if the receiver is (strictly) less than all elements in the argument. The method should *assume* the argument:

   - is *enumerable* (in the sense of the `Enumerable` mixin)
   - contains only items that can be *compared* (in the sense of the `Comparable` mixin) to the receiver.

Solution:

(a) Many solutions possible, such as:

```
module Comparable
  def between (x,y)
    x < self && self < y # many other ways of course
  end
end
```

(b) Many solutions possible, such as:

```
module Comparable
  def less_than_all? e
    e.all? {|i| self < i }
  end
end
```

Name:_____

8. (**18** points) (Function vs. object-oriented decomposition) Consider this SML code that implements its own lists with a few standard operations:

```
datatype 'a mylist = Empty | NonEmpty of 'a * ('a mylist)

exception Bad of string

fun head xs =
    case xs of
       Empty => raise (Bad "head of empty list")
     | NonEmpty(x,_) => x

fun tail xs =
    case xs of
       Empty => raise (Bad "tail of empty list")
     | NonEmpty(_,ys) => ys

fun length xs =
    case xs of
       Empty => 0
     | NonEmpty(_,ys) =>  1 + length ys

fun is_longer (xs,ys) = length xs > length ys

fun append (xs,ys) =
    case xs of
       Empty => ys
     | NonEmpty(x,zs) => NonEmpty(x, append(zs,ys))
```

(a) Write an SML function `sum` of type `int mylist -> int` that produces the sum of all the elements in the argument.

Part (b) of this question is on the next page.

Name:_____

(b) Port the SML code, including `sum`, to Ruby, converting the code to use good object-oriented style but otherwise using the same algorithms. Your code should follow all these guidelines:

- Have 3 classes: `Mylist` and two subclasses of it, `Empty` and `NonEmpty`.
- One of your classes should have an `initialize` method.
- Your code should have no mutation (so aliasing versus copying is not an issue).
- You should not use `nil`, `is_a?`, or any standard-library classes (e.g., `Array`).
- Do not define an exception `Bad`; in Ruby, `raise` can just take a string directly for an error message.

The sample solution is 40–45 lines but every line is short and many are `end`.

Solution:

(a)
```
fun sum xs =
    case xs of
       Empty => 0
     | NonEmpty(x,ys) => x + sum ys
```

(b)
```
class Mylist
  def is_longer ys
    length > ys.length
  end
end
```

```
class Empty < Mylist
  def head
    raise "head of empty list"
  end
  def tail
    raise "tail of empty list"
  end
  def length
    0
  end
  def sum
    0
  end
  def append ys
    ys
  end
end
```

```
class NonEmpty < Mylist
  def initialize (h,t)
    @h = h
    @t = t
  end
  def head
    @h
  end
  def tail
    @t
  end
  def length
    1 + @t.length
  end
  def sum
    @h + @t.sum
  end
  def append ys
    NonEmpty.new(@h,@t.append(ys))
  end
end
```

Name:_____

9. (**13** points) (Subtyping) This problem considers a language like in lecture containing (1) records with mutable fields, (2) 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.

   Consider these nine type definitions:

```
type t1 = { h : int}
type t2 = { h : int, f : int}
type t3 = { f : int, g : {a : int}, h : int}
type t4 = { f : int, g : {a : int, b: int}, h : int}
type t5 = { f : int, g : {a : int, b: int, c: int}, h : int}

type t6 = { h : int} -> { h : int}
type t7 = { h : int} -> { f : int, g : {a : int}, h : int}
type t8 = { f : int, g : {a : int}, h : int} -> { h : int}
type t9 = { f : int, g : {a : int}, h : int} -> { f : int, g : {a : int}, h : int}
```

   (a) Which of the nine types above are subtypes of `t1`?
   Solution:
   t1, t2, t3, t4, t5

   (b) Which of the nine types above are subtypes of `t2`?
   Solution:
   t2, t3, t4, t5

   (c) Which of the nine types above are subtypes of `t3`?
   Solution:
   t3

   (d) Which of the nine types above are subtypes of `t4`?
   Solution:
   t4

   (e) Which of the nine types above are subtypes of `t5`?
   Solution:
   t5

   (f) Which of the nine types above are subtypes of `t6`?
   Solution:
   t6, t7

   (g) Which of the nine types above are subtypes of `t7`?
   Solution:
   t7

   (h) Which of the nine types above are subtypes of `t8`?
   Solution:
   t6, t7, t8, t9

   (i) Which of the nine types above are subtypes of `t9`?
   Solution:
   t7, t9

Name:_____

This is an extra sheet of paper in case you need more room for any of the questions. If you use it, please write "see extra page" on the page with the question. In either case, please do NOT remove it from your exam.