Name: _____ Student ID #: _____

CSE 341 Summer 2019 Final Exam

August 23, 2019

Do not turn the page until you are instructed to do so.

Rules/Guidelines:

- You must stop working **promptly** when time is called at 1:00pm. Any modifications to your exam (writing *or* erasing) will result in a penalty.
- This exam is closed-book, closed-note, and closed-device with the exception of *both sides* of *one* 8.5x 11" piece of paper.
- There are **85** points distributed unevenly among **4** multi-part questions.
- The exam is printed double-sided, with **11** numbed pages.
- If you abandon one answer and write another, *clearly cross out* the answer(s) you do not want graded and *draw a circle or box around* the answer you do want graded. When in doubt, we will grade a circled/boxed answer, or the answer that appears nearest to the question text if no answer is circled/boxed.
- If you write an answer on scratch paper, please both *clearly label* which question you are answering on the scratch paper and also *clearly indicate* on the question page that your answer is on scratch paper. Staple all scratch paper to the *end* of the exam before turning in.

Advice:

- Read all questions carefully. Be sure you understand the question *before* you begin your answer.
- The questions are not necessarily in order of difficulty. Feel free to skip around. Be sure you are able to at least attempt every question.
- Work on easier questions first, including the easiest parts of each question. Do not feel the need to answer all parts of one question before moving on to another. You can always circle back.
- Write clearly and legibly. We cannot award credit for answers we cannot read.
- Write down any thoughts or intermediate steps so we can award partial credit, but be sure to clearly indicate your final answer.
- If you have questions, raise your hand to ask. The worst that can happen is we will say "I can't answer that."
- Ask questions as soon as you have them. Do not wait until you have several questions at once.
- Relax. You are here to learn.

Point Distribution:

Problem	*1*	*2*	*3*	*4*	*Total*
Points	22	18	19	26	85

Name: _____ Student ID #: _____

1. (Racket Programming – 22 points)

a) Write a Racket function `flatten` that takes a list as an argument and returns another list. The result list should contain no sub-lists (that is, no elements that are themselves lists), but should otherwise contain all the same values in the same order as the argument list. For example, `(flatten (list 1 2 (list 3 4 5) 6))` should evaluate to `'(1 2 3 4 5 6)`. You may assume the argument list does not contain any cons cells that are not lists and does not contain any structs.

```
(define (flatten xs)
  (cond [(null? xs) null]
        [(null? (car xs)) (flatten (cdr xs))]
        [(list? (car xs)) (flatten (cons (caar xs) (cons (cdar xs) (cdr xs))))]
        [#t (cons (car xs) (flatten (cdr xs)))]))
```

b) Consider the following function, assuming `flatten` works as described above:

```
(define (mystery xs)
  (cond [(null? xs) xs]
        [(list? (car xs)) (cons (flatten (car xs)) (mystery (cdr xs)))]
        [#t (cons (car xs) (mystery (cdr xs)))]))
```

What do each of the following calls evaluate to?

i. `(mystery (list 1 2 (list 3 4 5) 6))`

`'(1 2 (3 4 5) 6)`

ii. `(mystery (list (list 1 2) (list 3 4 (list 5 6))))`

`'((1 2) (3 4 5 6))`

iii. `(mystery (list (list 1 (list 2 3) (list 4 5 6))))`

`'((1 2 3 4 5 6))`

iv. `(mystery (list (list 1 (list 2 null (list 3)) (list 4 5 6))))`

`'((1 2 3 4 5 6))`

c) Give an example of a non-null list `xs` such that `(flatten xs)` and `(mystery xs)` evaluate to the same result.

Any list with no sublists.

d) Give an example of a non-null list `ys` such that `(flatten ys)` and `(mystery ys)` DO NOT evaluate to the same result.

Any list with at least one sublist.

Name: _____ Student ID #: _____

2. (Thunks and Streams – 18 points) 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. We assume all streams are pure (no printing, mutation, etc.). Assume the following streams are defined:

```
nats = 1, 2, 3, 4, 5, ...  (the natural numbers)
evens = 2, 4, 6, 8, 10, ...  (the positive even integers)
negs = -1, -2, -3, -4, -5, ...  (the negative integers)
```

a) Write a Racket function `weave-streams` that takes two stream arguments, `s1` and `s2`, and returns a stream. The resulting stream should contain alternating elements from the two argument streams. That is, the odd-numbered elements of the result stream should be elements (in order) from `s1`, and the even-numbered elements of the result stream should be elements (in order) from `s2`.

For example, `(weave-streams nats negs)` would represent `1, -1, 2, -2, 3, -3, ...`.

```
(define (weave-streams s1 s2)
  (letrec ([loop (lambda (curr next)
                   (lambda () (cons (car (curr))
                                    (weave-streams next (cdr (curr))))))])
    (loop s1 s2)))
```

b) Consider the following Racket function:

```
(define (every-other s)
  (letrec ([loop (lambda (s)
                   (lambda () (cons (car (s)) (loop (cdr ((cdr (s))))))))])
    (loop s)))
```

i. What does `(every-other nats)` evaluate to? You may either describe the result in 1 precise English sentence or give a clear, unambiguous numerical representation of the values in the result.

The positive odd integers (1, 3, 5, 7, …)

ii. Using only the streams above and standard Racket functions, fill in the blank so that `evens2` is bound to a stream representing the same values as `evens`?

```
(define evens2 (every-other ___(cdr (nats))_____))
(define evens2 (every-other ___(weave-streams evens evens)_____))
```

c) Use `every-other` to write a Racket function `split-stream` that takes a stream as an argument and returns a pair of streams such that the first stream in the result pair contains the 1st, 3rd, 5th, etc. of the argument stream and the second stream in the result pair contains the 2nd, 4th, 6th elements of the argument stream.

For example, `(split-stream evens)` would return the pair `(s1 . s2)` where `s1` represents the values `2, 6, 10, 14, ...` and `s2` represents the values `4, 8, 12, 16, ...`.

```
(define (split-stream s)
  (cons (every-other s) (every-other (cdr (s)))))
```

Name: _____ Student ID #: _____

3. (Interpreter Implementation – 19 points) 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 mnot (e) #:transparent)
```

It has the following semantics:

a) The lone subexpression is evaluated exactly once.
b) If the subexpression evaluates to the integer 0, the `mnot` evaluates to the integer 1.
c) If the subexpression evaluates to any integer that is not 0, the `mnot` evaluates to the integer 0.
d) If the subexpression evaluates to something other than an integer, evaluating the `mnot` raises an error.

Name: _____ Student ID #: _____

a) Consider this *incorrect* implementation of `mnot` in "the big cond" of `eval-under-env`:

```
[(mnot? e)
 (let ([v (eval-under-env (mnot-e e) env)])
   (if (= (int-num v) 0)
       (int 1)
       (int 0)))]
```

For each of the following expressions, determine which of the following will occur:

A. The interpreter will produce the *correct result*
B. The interpreter will produce an *incorrect result*
C. The interpreter will raise a MUPL error
D. The interpreter will raise a Racket exception
E. The interpreter will not terminate

`(eval-exp (mnot (int 0)))`	**A** B C D E
`(eval-exp (mnot (add (int 1) (int 0))))`	**A** B C D E
`(eval-exp (mlet "p" (apair (int 0) (int 1)) (mnot (var "p"))))`	A B C **D** E
`(eval-exp (mlet "p" (apair (int 1) (int 0)) (mnot (first (var "p")))))`	**A** B C D E

`(eval-exp (mlet "x" (int -4)`
`(mlet "y" (int 3)`
`(ifnz (mnot (add (var "x") (var "y")))`
`(int 3)`
`(int 4)))))`
	A B C D E

b) Repeat part (a) for this *incorrect* implementation of `mnot`:

```
[(mnot? e)
 (let ([v (eval-under-env (mnot-e e) env)])
   (if (int? v)
       (if (> (int-num v) 0)
           (int 0)
           (int 1))
       (error "MUPL mnot applied to non-int")))]
```

`(eval-exp (mnot (int 0)))`	**A** B C D E
`(eval-exp (mnot (add (int 1) (int 0))))`	**A** B C D E
`(eval-exp (mlet "p" (apair (int 0) (int 1)) (mnot (var "p"))))`	A B **C** D E
`(eval-exp (mlet "p" (apair (int 1) (int 0)) (mnot (first (var "p")))))`	**A** B C D E

`(eval-exp (mlet "x" (int -4)`
`(mlet "y" (int 3)`
`(ifnz (mnot (add (var "x") (var "y")))`
`(int 3)`
`(int 4)))))`
	A **B** C D E

c) Write a *correct* implementation of the `mnot` case in the "the big cond":

```
[(mnot? e)
 (let ([v (eval-under-env (mnot-e e) env)])
   (if (int? v)
       (if (= (int-num v) 0)
           (int 1)
           (int 0))
       (error "MUPL mnot applied to non-int")))]
```

d) Now suppose we did not implement the `mnot` expression in the MUPL interpreter. Write a Racket function `mnot-macro` that acts as MUPL macro and simulates the behavior of `mnot`. Your function should take a MUPL expression `e` as an argument and produce a new MUPL expression that is equivalent to `(mnot e)`, but would evaluate in the original MUPL interpreter.

```
(define (mnot-macro e)
  (ifnz e (int 0) (int 1)))
```

Name: _____ Student ID #: _____

4. (Functional v. object-oriented decomposition – 26 points) Consider the following SML code:

```
datatype Train = CargoCar of string
               | PassengerCar of int
               | MultiCar of Train * Train

fun toString t =
    case t of
CargoCar _ => "Cargo"
      | PassengerCar _ => "Passengers"
      | MultiCar(t1, t2) => toString t1 ^ ":-:" ^ toString t2

fun numPassengers t =
    case t of
     PassengerCar n => n
      | MultiCar (t1, t2) => (numPassengers t1) + (numPassengers t2)
      | _ => 0

fun numCars t =
    case t of
     MultiCar (t1, t2) => (numCars t1) + (numCars t2)
      | _ => 1
```

a) Port the SML code to equivalent Ruby code using good object-oriented style. In particular, follow these guidelines:

- Define four classes: a superclass `Train` and three subclasses `CargoCar`, `PassengerCar`, and `MultiCar`.
- Define an `initialize` method in three of the five classes.
- Do not have any unnecessary code duplication.
- Do not use mutation, `is_a?`, or any standard library classes.

Our solution is approximately 45-50 lines, but all lines are short and many consist only of `end`. Write your code on the next page.

Name: _____ Student ID #: _____

Write your solution to part (a) here:

```
class Train
  def numPassengers
    0
  end

  def numCars
    1
  end
end


class CargoCar < Train
  def initialize (s)
    @cargo = s
  end

  def toString
    "Cargo"
  end
end
```

```
class PassengerCar < Train
  def initialize (n)
    @pass = n
  end

  def toString
    "Passengers"
  end

  def numPassengers
    @pass
  end
end


class MultiCar < Train
  def initialize(t1, t2)
    @t1 = t1
    @t2 = t2
  end

  def toString
    @t1.toString + ":-:" + @t2.toString
  end

  def numPassengers
    @t1.numPassengers + @t2.numPassengers
  end

  def numCars
    @t1.numCars + @t2.numCars
  end
end
```

Name: _____ Student ID #: _____

`

b) Add one or more method definitions (including indicating to which class(es) you are adding them) so that `Train` and all its subclasses can be compared using the operators defined in the `Comparable` mixin. `Train`s should be compared based on number of cars; that is, a `Train` with fewer cars is considered "less than" a `Train` with more cars.

```
class Train
  include Comparable

  def <=> other
      numCars <=> other.numCars
  end
end
```

c) Consider the following added code, which includes the `Enumerable` mixin on `Train` and its subclasses:

```
class Train
  include Enumerable
  def each
    yield self
  end
end
```

```
class MultiCar < Train
  include Enumerable
  def each
    @t1.each {|c| yield c}
    @t2.each {|c| yield c}
  end
end
```

Recall that the `Enumerable` mixin uses `each` to implement the method `count`, which takes a block argument and produces the number of elements for which the block produces true.

Write a *single line* of Ruby code that will produce the number of cars in a `Train t` with a positive number of passengers. You may not use `is_a?` in your solution.

```
t.count { |c| c.numPassengers > 0 }
```