Name: ______________________________ netid : ______________________

CSE 341 Winter 2019 Final

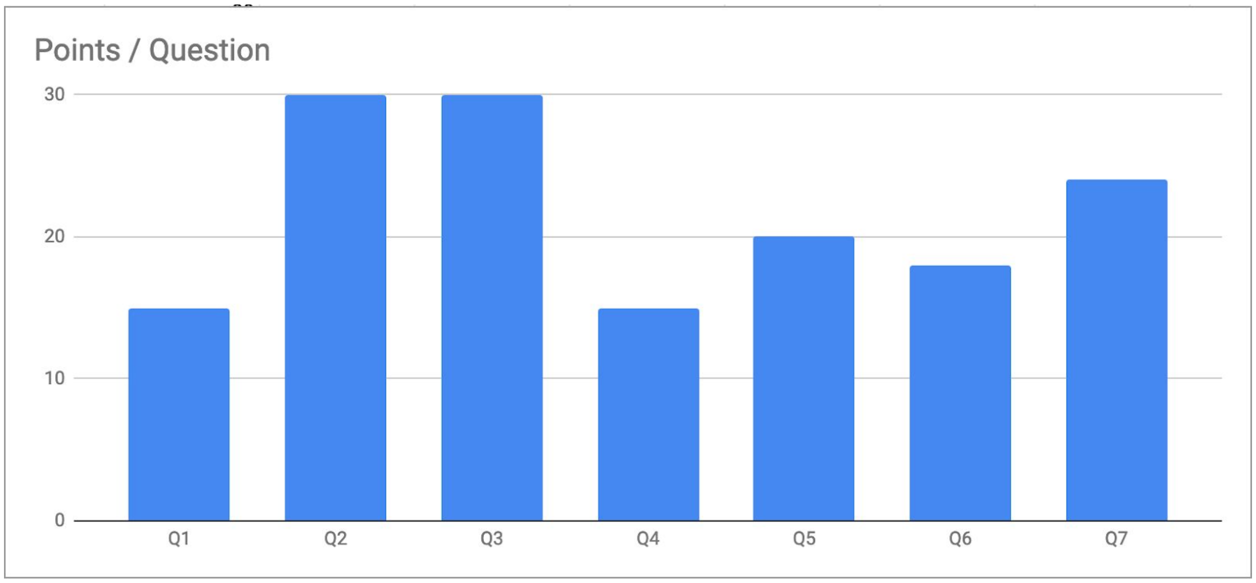
Please do not turn the page until 2:30.

Rules:

- The exam is closed-book, closed-note, etc. except *both sides* of a 8.5x11in page.
- **Please stop promptly at 4:20.**
- There are **152 points**, distributed **unevenly** among 7 multi-part questions.
- ***QUESTIONS VARY GREATLY IN DIFFICULTY. GET EASY POINTS FIRST!!!***
- The exam is printed double-sided, with pages numbered up to 23.

Advice:

- Read the questions carefully. Understand before you answer.
- Write down thoughts and intermediate steps so we can give partial credit.
- **Clearly indicate your final answer.**
- **WRITE CLEARLY. No partial credit for anything we can't read.**
- Questions are not in order of difficulty. **Answer everything.**
- If you have questions, ask.
- Relax. You are here to learn.



Name: ______________________________ netid : ______________________

QUESTION 1 (15 points) *(Racket Programming)*

(A) What does the following program print?

```
; Note: println prints to the console, like in Java
(define x 1)
(define y 341)

(define f
  (let ([y x])
    (begin (println y)
           (lambda (z)
             (begin (set! x (+ x z))
                    x)))))

(println (f 1))
(println (f 2))
(println (f 3))

; write output below
```

Solution

1
2
4
7

Name: ______________________________ netid : _______________________

(B) What is ans bound to after the following program executes?

```
(define (split l)
  (letrec ([loop (lambda (xs ys zs)
                   (if (null? xs)
                       (cons ys zs)
                       (loop (cdr xs) zs (cons (car xs) ys))))])
    (loop l null null)))

(define ans (car (split (list 1 2 3 4))))
```

ans = ________**'(3 1)**__________________________________

(C) What does the underlined expression evaluate to?

```
(define-syntax binary-search
  (syntax-rules ()
    [(binary-search (node left right))
     (struct left (node right x))]))

(binary-search (+ x y))

(define y (x 1 2 3))
(+ (x-+ y) (x-x y))
```

Result of underlined expr = _____________**4**______________________

Name: ________________________________ netid : ________________________

QUESTION 2 (30 points) *(Streams)*

Recall that a stream is a thunk that returns a pair where the `cdr` is a stream.

(A) Write a Racket function `incrementor` which takes two arguments, `x` and `f`. You can assume `f` is a function which takes two arguments. `incrementor` should return a stream where the `n`th value (starting at 1) is the result of calling `f` with first argument `x` and second argument `n`. *Our sample solution is 4 lines.*

For example:

```
(incrementor 3 (lambda (x n) (+ x n)))
```

Should return a stream whose first five values will be: `4, 5, 6, 7, 8`

```
(define (incrementor x f)
  (define (helper n)
    (lambda () (cons (f x n) (helper (+ n 1)))))
  (helper 1)

)
```

Name: ________________________________ netid : ________________________

(B) Write a Racket function `slicer` whose first argument `s` is assumed to be a stream and whose second argument `n` is assumed to be a positive integer (> 0). It should return a stream that consists of every `n`th element of `s`. *Our sample solution is 7 lines.*

For example:

```
(slicer (incrementor 3 (lambda (x n) (+ x n))) 2)
```

Should return a stream whose first three values are: `5, 7, 9`

It does not matter when you choose to evaluate the passed in stream `s`, so long as the resulting stream is correct.

```
(define (slicer s n)
   (define (helper s x)
     (if (= x 1)
     (lambda () (cons (car (s)) (helper (cdr (s)) n)))
   (helper (cdr (s)) (- x 1))))
  (helper s n)

)
```

Name: ______________________________ netid : ______________________

(C) Define `multiple-streams` to be a "stream of streams" where the `n`th stream (starting at 1) of `multiple-streams` is another stream containing every multiple of `n`, starting at `n`.

Use only `incrementor` and `slicer` from parts (A) and (B) above, numbers, variables, `+,` and `lambda`. *Our sample solution is 4 lines.*

For example, the first element of `multiple-streams` should be a stream of all multiples of 1:

1, 2, 3, 4, 5, …

The second element of `multiple-streams` should be a stream of all multiples of 2:

2, 4, 6, 8, 10, …

```
(define multiple-streams
  (incrementor
    (incrementor 0 (lambda (x n) (+ x n)))
    slicer)








)
```

Name: ________________________________ netid : ________________________

(D) After evaluating the following code, assuming parts (A - C) work correctly, what is `foo` bound to?

```
(define foo
  (* (car ((cdr ((cdr ((car ((cdr (multiple-streams))))))))))
     (car ((cdr ((car ((cdr ((cdr (multiple-streams)))))))))))
```

`foo =` **36**

Name: ________________________________ netid : ________________________

QUESTION 3 (30 points) *(MUPL Interpreter)*

Consider the following subset of the MUPL interpreter from homework. In this question we will consider a new feature: a `subtract` struct for subtracting two `int` expressions.

```
(struct var       (string))     ;; a variable, e.g., (var "foo")
(struct int       (num))        ;; a constant number, e.g., (int 17)
(struct add       (e1 e2))      ;; add two expressions (e1 + e2)
(struct ifnz      (e1 e2 e3))   ;; if not zero e1 then e2 else e3
(struct mlet      (var e body)) ;; a local binding (let var = e in body)
(struct subtract  (e1 e2))     ;; subtract e2 from e1 (e1 - e2)

(define (envlookup env str) ...)

(define (eval-under-env e env)
  (cond [(var? e)      ...]
        [(int? e)      ...]
        [(add? e)      ...]
        [(ifnz? e)     ...]
        [(mlet? e)     ...]
        [(subtract? e) ...]
        [#t (error (format "bad MUPL expression: ~v" e))]))
 (define (eval-exp e) (eval-under-env e null))
```

- An `int` evaluates to itself and a `var` evaluates to the value it is bound to in the environment.

- An `add` evaluates its subexpressions and, assuming they evaluate to `int`s, produces the `int` that is their sum. Gives the error "`MUPL addition applied to non-number`" if not given two `int`s.

- For `(ifnz e1 e2 e3), e1` is first evaluated to a value `v`. If `v` is an `int` not equal to 0, then the result is evaluating `e2`. If `v` is 0, then then the result is evaluating e3. If `v` does not evaluate to an `int`, then gives the error "`MUPL ifnz applied to non-number`".

- An `mlet` evaluates its first subexpression to a value `v`, then evaluates the second subexpression in an environment extended to map the given name to `v`.

- A `subtract` is just like an `add`, but it subtracts instead of adding.

- Interpreting anything else gives the error "`bad MUPL expression: ~v`" where "`~v`" is replaced by whatever was passed into the interpreter.

Name: ______________________________ netid : ________________________

For each sub-problem, consider a buggy `subtract` implementation, and give the result bound to `foo` after evaluating this:

```
      (define foo (eval-exp (subtract (add (int 15) (int 8)) (int 26))))
```

(Write the MUPL error message from the previous page if it raises an error in the interpreter, "exception" if it raises a Racket exception, and "does not halt" if it theoretically runs forever)

```
a) [(subtract? e) (let ([v1 (eval-under-env (subtract-e1 e) env)]
                        [v2 (subtract-e2 e)])
                       (eval-under-env (add v1 (- (int-num v2))) env))]
```

Result: **bad MUPL expression: -26**
Note that when "-26" is passed into add, it is evaluated, which is why we get this instead of the error for non-int in add.

```
b) [(subtract? e) (let ([v1 (subtract-e1 e)]
                        [v2 (eval-under-env (subtract-e2 e) env)])
                       (int (- (int-num (eval-under-env (add v1 v2) env)))))]
```

Result: **(int -49)**

```
c) [(subtract? e) (eval-under-env
                     (add (subtract-e1 e)
                          (subtract (int 0) (subtract-e2 e))) env)]
```

Result: **Does not halt**

Name: ______________________________ netid : ______________________

```
d) [(subtract? e) (let ([v1 (subtract-e1 e)] [v2 (subtract-e2 e)])
                   (eval-under-env
                     (ifnz v2
                           (add (int (- 1))
                                (subtract v1 (add (int (- 1)) v2)))
                           v1) env))]
```

Result: **(int -3)**

We also accepted "Does Not Halt" because the wording of the question was misleading. Theoretically, this does run forever, just not with the provided input. The question made it sound like if it could, for some input, run forever, you should write "does not halt."

```
e) [(subtract? e) (eval-under-env
                   (mlet "a" (subtract-e1 e)
                     (mlet "b" (subtract-e2 e)
                       (int (- (int-num (var "a"))
                               (int-num (var "b")))))) env)]
```

Result: **exception** **(Specifically we try to get the int-num for a var struct.)**

f) Write a `subtract` branch for the interpreter that works as described above. Do not worry about giving an error if not given two `int`s.

```
[(subtract? e)
   (let ([v1 (eval-under-env (add-e1 e) env)]
         [v2 (eval-under-env (add-e2 e) env)])
          (int (- (int-num v1) (int-num v2))))
]
```

Name: ______________________________ netid : ______________________

QUESTION 4 (15 points) *(Ruby Subclasses)*

Consider a simple Calculator class which stores the “current result” in instance variable @val and supports addition and subtraction:

```
class Calculator
   attr_accessor :val
   def initialize(val)
       @val = val
   end
   def add(val)
       @val = @val + val
       @val
   end
   def subtract(val)
       @val = @val - val
       @val
   end
end
```

On the following page, implement a subclass CalculatorUndo which provides an `undo` method. `undo` should return the current value of `@val` and restore `@val` to its previous value. You may ignore the case that calls `undo` when there are no operations to undo. Please follow good OOP style and use calls to `super` as appropriate.

For example:

```
c = CalculatorUndo.new(5) # initially, @val = 5
c.add(4) # now @val = 9
c.subtract(7) # now @val = 2
c.add(9) # now @val = 11
c.undo # returns 11, now @val = 2
c.undo # returns 2, now @val = 9
```

Name: ______________________________ netid : ______________________

```
class CalculatorUndo < Calculator

def initialize(val)
  super
  @history = Array.new
end

def add(val)
  @history.push @val
  super
end

def subtract(val)
  @history.push @val
  super
end

def undo
  old = @val
  @val = @history.pop
  old
end

end
```

Name: ______________________________ netid : ______________________

QUESTION 5 (20 points) *(OOP, Mixins, and Porting)*

A *path* is a sequence of moves. Consider the following Ruby code to represent single-move paths in directions East, West, North, and South as well as multi-move paths (ComboPath) which append two paths:

```
class Path
  def deltaX
    0
  end
  def deltaY
    0
  end
  def deltaXY
    [deltaX, deltaY]
  end
end

class E < Path
  def deltaX
    1
  end
end

class W < Path
  def deltaX
    -1
  end
end
```

```
class N < Path
  def deltaY
    1
  end
end

class S < Path
  def deltaY
    -1
  end
end

class ComboPath < Path
  def initialize (p1, p2)
    @p1 = p1
    @p2 = p2
  end
  def deltaX
    @p1.deltaX + @p2.deltaX
  end
  def deltaY
    @p1.deltaY + @p2.deltaY
  end
end
```

```
p = ComboPath.new( W.new,
      ComboPath.new( W.new,
        ComboPath.new( N.new,
          ComboPath.new( N.new, E.new ))))

pos = p.deltaXY;
```

(A) What is `pos` bound to in the code above?

`[-1, 2]`

Name: ______________________________ netid : ______________________

As we saw in lecture, Ruby's `Enumerable` mixin adds many useful methods in terms of the underlying class's `each` method. `each` takes no regular arguments and a block that takes a single argument.

The code below adds `each` method definitions for `Path` and `ComboPath` objects, ensuring that if `p` is an instance of `Path`, then `p.each` calls its block argument on all the single-move paths in `p` in order. (In order!)

```
class Path
  include Enumerable

  def each
    yield self
  end
end
```

```
class ComboPath
  def each
    @p1.each {|p| yield p }
    @p2.each {|p| yield p }
  end
end
```

(B) Use `each` to implement a `Path` method `allPrefixes` which produces an array of all the prefixes of a path. For example, given the definition of `p` above:

```
p.allPrefixes
```

Should return an array with all the prefixes of `p`, i.e., an array equivalent to:

```
[ W.new
, ComboPath.new( W.new, W.new )
, ComboPath.new( W.new,
    ComboPath.new( W.new, N.new ))
, ComboPath.new( W.new,
    ComboPath.new( W.new,
      ComboPath.new( N.new, N.new )))
, ComboPath.new( W.new,
    ComboPath.new( W.new,
      ComboPath.new( N.new,
        ComboPath.new( N.new, E.new ))))
]
```

Name: ______________________________ netid : ______________________

Remember that Ruby arrays provide methods like `size` to get the number of elements, `push` to add an element to the end of an array, and indexing from -1 to get the last element of an array. *Our sample solution is 10 lines.*

```
class Path
  def allPrefixes
    paths = []
    each do |p|
      if paths.size == 0 then
        paths.push p
      else
        q = paths[-1]
        paths.push (ComboPath.new(q, p))
      end
    end
    paths
  end
end
```

Name: ________________________________ netid : ________________________

(C) Add method definitions (indicate what classes you are adding them to—it can be a single method added to a single class) such that if `p` is an instance of any subclass of `Path`, then `p.furthestWest` returns the smallest "deltaX" value reached by any prefix of `p`. Note that for the definition of `p` above, `p.furthestWest` should return -2.

```
class Path

  def furthestWest
    min = nil
    allPrefixes.each do |p|
      if min.nil?
        min = p.deltaX
      else
        min = [min, p.deltaX].min
      end
    end
    min
  end

  # alternate
  def furthestWest
    allPrefixes.map {|p| p.deltaX}.min
  end
end
```

Name: ________________________________ netid : ________________________

QUESTION 6 (18 points) *(Type Systems)*

A type system is **sound** if it accepts (can type check) *only* programs that will never have any runtime type mismatch errors. That is, a sound type system may reject programs which are actually safe.

Conversely, a type system is **complete** if *all* programs that never have any runtime type mismatch errors are always accepted. That is, a complete type system may accept programs that are not safe.

For each part below, indicate whether the proposed type system is Sound, Complete, Both, or Neither. You do not need to explain your answers.

(a) A type system that rejects all programs

Sound. If a type system rejects all programs, then it won't accept any unsafe programs

(b) A type system that accepts all programs

Complete. As a converse to (a), if you accept all programs you accept all safe programs

(c) Java's type system extended to allow any two types to be subtypes of each other

Complete. This is actually the same as (b), because of the rule that if A is a subtype of B and e has type A, then e has type B. Then since anything is a subtype of anything else, this means all expressions have all types. When you're dealing with subtyping, types aren't unique!

(d) A type system for Racket which rejects `(+ 1 (list 1))` but accepts all other programs

Complete. Racket already accepts all bug-free programs, and excluding a buggy program doesn't change that.

Name: ______________________________ netid : ______________________

(e) SML's type system extended so that `1 + (if false then [1] else 1)` has type int

Sound. As a converse to the above, allowing SML to accept a program without any bugs will preserve soundness

(f) Consider a small Racket-like programming language with `+`, `if`, `#t`, `#f`, integer constants, and one-argument lambdas. Its type system has the following rules:

(i) The (only) types are **bool**, **int**, and **fun**

(ii) `#t` and `#f` have type **bool**

(iii) Any integer constant has type **int**

(iv) If `e1` and `e2` have type **int**, then the expression `(+ e1 e2)` has type **int**

(v) If `e1` has type **bool** and there's some type *T* such that `e2` and e3 both have type *T*, then the expression `(if e1 e2 e3)` has type *T*.

(vi) The expression `(lambda (`*x*`) e)` has type **fun** for any variable *x*.

(vii) If `f` has type **fun** and there's some type *T* such that `e` has type *T*, then `(f e)` has type *T*.

(viii) All programs which can't be typed by the above rules are rejected.

Neither.
This language is incomplete since it rejects (+ 1 (if #t 1 #t)), which will still execute without bugs.
It's unsound because of the function application typing rule: the function (lambda (x) #t) has type fun, and 1 has type int, so ((lambda (x) #t) 1) has type int. But then (+ 1 ((lambda (x) #t) 1)) will be given type int (and so accepted) even though it evaluates to (+ 1 #t), which is an error.

Name: ______________________________ netid : ______________________

QUESTION 7 (24 points) *(Subtyping)*

Consider a language like in lecture containing (1) records with mutable fields, (2) higher-order functions, and (3) subtyping.

Recall that a subtyping relationship is sound if 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.

(A) Indicate T if the proposed subtyping is sound, otherwise indicate F. You do not need to explain your answers.

		T / F

a	`{f2 : string}`
	is a subtype of
	`{}`
		T

b	`{f1 : string, f2 : {g1 : string, g2 : int} }`
	is a subtype of
	`{f1 : string, f2 : {g2 : int, g1 : string} }`
		T

c	`string -> int` is a subtype of `string -> int`	**T**

d	`{f2 : {g1: int}} -> string`
	is a subtype of
	`{f2 : {g1: int}, f3 : string} -> string`
		T

e	`{f1 : int} -> {f1 : string, f2 : {g1 : int}}`
	is a subtype of
	`{f1 : int, f3 : int} ->`
	`  {f1 : string, f2 : {g1 : int, g2 : string}}`
		F

Name: ______________________________ netid : ______________________

f	`{f1 : int} -> {f1 : string, f2 : {g1 : int, g2: string}}` is a subtype of `{f1 : int, g3: string} -> {f1 : string, f2 : {g1 : int}}`	F

(B) Assume we change the language so that only fields of type `int` and `string` are mutable, i.e., it is impossible to change the value of fields containing records.

Which, of your answers to part A change (check any that change)?

	Change?
a	NO
b	NO
c	NO
d	NO
e	NO
f	YES

Name: ______________________________ netid : ______________________

Extra Credit

EC1)
Consider the following code

```
(struct sml (functional style) #:transparent #:mutable)

(define (typecheck! x)
  (if (null? x)
      null
      (let ([h (car x)])
        (begin
          (set-sml-style! h (- 2 (sml-style h)))
          (cons h (typecheck! (cdr x)))))))
```

Fill in the blank so that all calls to `equal?` in the following program will return `#t`

```
(define djg
```

```
(let ([x (sml 0 0)]) (list x x))
; a student also came up with the following
; (begin (set! typecheck! (lambda (x) x))
;        (list (sml 0 0) (sml 0 0)))
```

```
)

(equal? (sml-style (car djg)) 0)
(equal? (sml-style (car (cdr djg))) 0)
(typecheck! djg)
(equal? (sml-style (car djg)) 0)
(equal? (sml-style (car (cdr djg))) 0)
```

Name: ______________________________ netid : _______________________

EC2) Fill in the blanks to make this program evaluate to `"racket"`

```
; S takes in function f and g and an argument x
; and applies the function (f x) to (g x)
(define S
  (lambda (f g x)
    ((f x) (g x))))

(define K
  (lambda (x)
    (lambda (y) x)))

(let ([a ____________(lambda (x) "racket")_____________________]
      [b ____________null______________________________________])
; you can also bind a to (lambda (x) x) and b to "racket"
  (S (K a)
     (K b)
     (S K K b)))
```

Name: ________________________________ netid : ________________________

EC3) What variables must be previously defined for the following program to run? Include no more variable names than necessary.

```
(define-syntax by
  (syntax-rules (sally)
    [(by a (t (sally) c)) (c a a t)]))

(define-syntax sea-shells
  (syntax-rules (the)
    [(sea-shells (the sea-shore) sally seas)
     (let ([sea-shore 2] [seas 5])
       (+ seas sally))]))

(by (the sea-shore)
    (sells (sally)
           sea-shells))
```

The only variable you need to define is the. After expanding the by macro, we get (sea-shells (the sea-shore) (the sea-shore) sells), and then expanding sea-shells gives us (let ([sea-shore 2] [sells 5]) (+ sells (the sea-shore)))