Rules:

- The exam is closed-book, closed-note, except for both sides of one 8.5x11 in piece of paper.
- Please stop promptly at 10:20.
- You can rip apart the pages, but please staple them back together before you leave.
- There are 100 points total, distributed unevenly among 8 questions (many with multiple parts).
- When writing code, style matters, but do not worry much about indentation.

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all the problems.
- If you have questions, ask.
- Relax. You are here to learn.
1. (13 points) This problem uses these Racket struct definitions to define a form of binary tree:

```
(struct leaf (data) #:transparent)
(struct node (data left right) #:transparent)
```

In trees made using these structs, note that:

- Leaves and internal nodes hold data in them. The data might be any Racket value.
- We assume the left and right fields of a value built with node are themselves binary trees.

Write a map function over these binary trees as follows:

- It should take two curried arguments (this is not the Racket default), first a function and second a tree. (You can get partial credit without currying.)
- It should return a new tree of the same shape as the tree argument but with the function argument applied to each tree element.
2. (10 points) This problem considers the difference between

\[
\text{(define foo (lambda (x) (let ([y e1]) e2))) ; Code A}
\]

and

\[
\text{(define foo (let ([y e1]) (lambda (x) e2))) ; Code B}
\]

Assume \(e1\) does not use \(x\).

(a) In roughly 2–4 English sentences, describe the general difference between “Code A” and “Code B.”

(b) Give an example \(e1\) and \(e2\) that make Code A and Code B not equivalent. You can provide additional code before the code above if you wish.
3. (13 points) Write a Racket function `twice-each` that takes a stream s and returns a stream. (Remember a stream is a thunk that returns a pair where the cdr is a stream.) The stream returned should be like s except each value generated by s is repeated twice. For example, if s generates 1, 2, 3, 4, ..., then (twice-each s) generates 1, 1, 2, 2, 3, 3, 4, 4, ...
4. (15 points) In languages with first-class functions and dynamic scope, we evaluate a function in the environment where it is called (extended to map the function’s parameter name to the result of evaluating the function’s argument at the call-site). Dynamic scope is a really bad idea, but it is not difficult to implement. It is easier to implement than lexical scope because we do not need closures: we can treat functions as values.

Below is a small language definition and (partial) interpreter for DUPL, which is like MUPL from our homework except:

- The language is smaller.
- Functions are not recursive: they are like \(\text{lambda} (x) \ e\) in Racket or \(\text{fn} x => e\) in ML.
- Function calls use dynamic scope.

```
(struct var (string) #:transparent)
(struct int (num) #:transparent)
(struct add (e1 e2) #:transparent)
(struct mylet (var e1 e2) #:transparent)
(struct fun (var body) #:transparent); var a Racket string, body a dupl expression
(struct call (e1 e2) #:transparent)
```

```
(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-dyn e env)
  (cond [(var? e) (envlookup env (var-string e))]
        [(int? e) e]
        [(fun? e) e]
        [(add? e) (let ([v1 (eval-dyn (add-e1 e) env)]
                         [v2 (eval-dyn (add-e2 e) env)])
                      (if (and (int? v1) (int? v2))
                          (int (+ (int-num v1) (int-num v2)))
                          (error "non-int in addition")))]
        [(mylet? e) (eval-dyn (mylet-e2 e)
                              (cons (cons (mylet-var e) (eval-dyn (mylet-e1 e) env))
                                    env))]
        [(call? e) ; your answer to part (a) would go here]
      )))
```

(a) Complete the interpreter by writing the case for \(\text{call}\). It should raise an error if the first argument does not evaluate to a function. Else it should call the function with the result of evaluating the argument. Use dynamic scope as described above.

(b) Complete this DUPL program such that:

- Evaluating it in an empty environment would produce \(\text{int} 17\).
- If we had lexical scope instead, then evaluation in an empty environment would cause an undefined-variable error.

```
(mylet "f" _________
  (mylet "x" _________
    (call _________ _________)))
```

The next page has room for your solutions.
Name:_______________________________

Put your solution to the problem on the previous page here.
5. (12 points) In this problem, we assume the purpose of the Java type system is to prevent “method missing” errors at run-time.

(a) For this purpose, is Java’s type system sound? Explain briefly, including a definition of soundness.

(b) For this purpose, is Java’s type system complete? Explain briefly, including a definition of completeness.

(c) Suppose we change the Java type system to disallow writing the constant `null`. Now is this revised type system sound? Explain briefly.

(d) Suppose we change the Java type system to disallow writing the constant `null`. Now is this revised type system complete? Explain briefly.
6. (12 points) This problem uses this Ruby code:

```ruby
class A
  def m1
    self.m2()
  end
  def m2
    puts "A-m2"
  end
end
module M
  def m3
    self.m4()
  end
end
class B < A
  def m2
    puts "B-m2"
  end
end
class C < A
  include M
  def m4
    puts "C-m4"
  end
end
class D < B
  include M
end
```

For each expression below, indicate “error” if evaluation would cause a method-missing error. If not, indicate what would be printed.

(a) B.new.m1
(b) B.new.m3
(c) C.new.m1
(d) C.new.m3
(e) D.new.m1
(f) D.new.m3
7. (10 points) Recall the Ruby method `is_a?` takes a class `c` as an argument and returns `true` if and only if the receiver is an instance of `c` or a subclass of `c`. Suppose `is_a?` was not provided by Ruby. Give an implementation of `is_a?` using a recursive helper method and the provided methods `class`, `superclass`, and `==`. That is, define the `is_a?` method. You can assume the argument to `is_a?` is a class.
8. (15 points) In this problem, we consider a language like in lecture containing (1) records with mutable fields, (2) higher-order functions, and (3) subtyping. Like in class, record subtyping includes width and permutation but not depth, and function subtyping allows contravariant arguments and covariant results. We also add type synonyms like in ML: the declaration `type t = ...` means `t` is equivalent to the type `...`.

For each of the following expressions (mostly just function calls), decide if the expression should type-check, answering “Yes” if it should type-check and “No” if it should not. If your answer is, “No,” give a possible implementation of the relevant functions so that the call would try to read a field of a record that does not exist or try to use addition on a non-number.

In your solutions, use the syntax `e.f` to read fields and `e1.f = e2` to write fields.

```plaintext
type int_pair = { car : int, cdr : int }
type pair_pair = { car : int_pair, cdr : int_pair }
type int_triple = { car : int, cdr : int, cgr : int }
type int_pair_fn = int_pair -> int_pair

val r1 = { cgr=5, cdr=6, car=9 }
val r2 = { car = { car = 1, cdr = 2}, cdr = { car = 3, cdr = 4 } }
val r3 = { car = r1, cdr = { car = 3, cdr = 4 } }
val r4 = { car = 7, cdr = 9 }

(* assume these variables are bound to functions with the given type *)
val f1 : int_pair -> int_pair = ...
val f2 : pair_pair -> int_pair = ...
val f3 : int_triple -> int_pair = ...
val f4 : { car : int } -> int_triple = ...
val f5 : int_pair_fn -> int_pair = ...
```

(a) `f1(r1)`
(b) `f1(r2)`
(c) `f2(r2)` followed by `r2.car.cdr`
(d) `f2(r3)` followed by `r3.car.cgr`
(e) `f3(r1)`
(f) `f3(r4)`
(g) `f5(f1)`
(h) `f5(f3)`
(i) `f5(f4)`
Name: ____________________________________________

More room for answers if needed.