Please do not turn the page until 8:30.

Rules:

- The exam is closed-book, closed-note, etc. except for two sides of one 8.5x11in piece of paper.
- Please stop promptly at 10:20.
- There are 126 points, distributed unevenly among 7 questions (most all with multiple parts):
- The exam is printed double-sided.
- Put your name on every page.

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.
- We will be scanning the exam to grade it. If you put an answer to a question in a non-intuitive spot, leave us a note. When in doubt, labeling where your answer is will never hurt.
- The questions are not necessarily in order of difficulty. Skip around! Make sure you get to all the questions, but do not do the exam in order.
- If you have questions, ask.
- Relax. You are here to learn.
1. (21 points) Suppose you want to define data structures a bit differently in Racket. Instead of just
cons cells, you want tcons (or triple cons) cells, with 3 parts.
You might accomplish this with a struct definition like so:

(struct tcons (fst snd trd) #:transparent)

(a) Suppose that a proper tlist made of tcons cells is just like a normal list made of cons cells,
except that fst and snd both hold list elements, and trd holds the rest of the list. Implement
tlist?, which returns true if and only if its argument is a proper tlist.
(b) Implement map-tlist, which will apply a given function to every element in a proper tlist, and
return a resulting tlist.
(c) Implement flatten, which will take any structure represented with possibly arbitrarily nested
tcons cells, and flatten it out. For instance:
(tcons (tcons 1 2 3) 4 (tcons 5 6 (tcons 7 8 9))) should be flattened out to a result
(list 1 2 3 4 5 6 7 8 9). Note that the result should be a list and not a tlist.

Solution:

(define (tlist? tl)
  (cond [(tcons? tl) (tlist? (tcons-trd tl))]
        [(null? tl) #t]
        [#t #f]))

(define (map-tlist f tl)
  (cond [(tcons? tl) (tcons (f (tcons-fst tl)) (f (tcons-snd tl)) (map-tlist f (tcons-trd tl)))]
        [(null? tl) '()]
        [#t #f]))

(define (flatten t)
  (cond [(tcons? t)
         (append
          (flatten (tcons-fst t))
          (flatten (tcons-snd t))
          (flatten (tcons-trd t)))]
        [#t (list t)]))
2. (8 points)
Consider the following definition in Racket:

(define (foo x y)
  (begin
    (print (mcar x))
    (print (mcar y))
    (set-mcar! x (+ (mcar x) 1))
    (set-mcar! y (- (mcar y) 1)))
    (print (mcar x))
    (print (mcar y))
)

Write a Racket expression which contains a call to foo, which results in all four print statements printing the number 2.

Solution:

(let ([x (mcons 2 2)]) (foo x x))
3. (Racket Streams) (23 points)

As in class, we define a stream to be a thunk which returns a pair when called, the cdr of which is itself a stream.

(a) Write a function sparse-stream which takes three arguments: default which is a single element, and two stream arguments elems and indices. sparse-stream should return a stream, which yields elements from elems in order, spaced out with repetitions of the default element. The number of default elements to insert before the next element from elems is given by the number retrieved from the indices stream.

For example, suppose the stream

\[ nats = 0,1,2,3,4,5 \ldots \]

is already defined. Then, \((\text{sparse-stream} -8 \ nats \ nats)\) should result in a stream which looks like:

\[ 0,1,-8,2,-8,-8,3,-8,-8,4,-8,-8,-8,-8,5,-8,-8,-8,-8,-8,-8,6 \ldots \]

(b) Write a function id-stream which takes one stream argument, and results in a stream which is identical to the argument. Use one call of sparse-stream to implement id-stream: do not simply return the argument.

(c) Suppose you have a result which you know was made by sparse-stream, but you’ve lost the original indices argument. Write a function recover-indices which takes two arguments: a default element and a stream s. Assuming that the stream s is the result of a call to sparse-stream, and that the original elems argument to the call to sparse-stream never yielded the default element, your result should be identical to the original indices argument to sparse-stream.

Solution:

\[
\text{(define (sparse-stream default elems indices)}
\]

\[
\text{ (letrec ([f (lambda (n els inds))}
\]

\[
\text{ (if (= n 0)}
\]

\[
\text{ (let ([elnext (els)]}
\]

\[
\text{ [indnext (inds)])}
\]

\[
\text{ (cons (car elnext)}
\]

\[
\text{ (lambda ()}
\]

\[
\text{ (f (car indnext) (cdr elnext) (cdr indnext)))})]
\]

\[
\text{ (cons default (lambda () (f (- n 1) els inds)))})])}
\]

\[
\text{ (lambda () (f 0 elems indices)))})]
\]

\[
\text{(define zeros (lambda () (cons 0 zeros)))}
\]

\[
\text{(define (id-stream s)}
\]

\[
\text{(sparse-stream -8 s zeros))}
\]

\[
\text{(define (recover-indices default s)}
\]

\[
\text{ (letrec ([f (lambda (s)}
\]

\[
\text{ (letrec ([g (lambda (n s)}
\]

\[
\text{ (let ([res (s)])}
\]

\[
\text{ (if (= default (car res))}
\]

\[
\text{ (g (+ n 1) (cdr res)))}
\]

\[
\text{ (cons n (lambda () (f (cdr res))))))])}
\]

\[
\text{ (g 0 s))])}
\]

\[
\text{ (lambda () (f (cdr (s))))})]
\]
4. **(21 points)** Below is the definition of TPL (Tiny Programming Language) abstract syntax. TPL is identical to MUPL, but with fewer expressions.

```
(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 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
```

(a) Implement an optimization function `precompute-add` which will take a TPL program, and return a TPL program which will always evaluate to the same answer, but has any addition expressions which consist of two constants replaced by their answer. For instance, the program `(add (int 3) (int 2))` should be replaced by `(int 5)` when optimized by `precompute-add`.

(b) Suppose you want to add a type system to TPL. Implement the simplest complete type checker for TPL (a type checker takes a TPL program and returns a bool). For type checking, `#t` means that the program typechecks, and `#f` means that it does not.

(c) Write an expression in TPL which, when evaluated, acts like stream, giving the natural numbers in order (i.e. 0,1,2,3,4...). Since we can’t call a function with 0 arguments, instead assume that a stream is called with `munit` in order to get the next result.

**Solution:**

```scheme
(define (precompute-add p)
  (cond 
    [(var? p) p]
    [(int? p) p]
    [(add? p)
      (let ([l (add-e1 p)]
                [r (add-e2 p)])
        (if (and (int? l) (int? r))
            (int (+ (int-num l) (int-num r)))
            (add (precompute-add l) (precompute-add r))))]
    [(fun? p) (fun (fun-nameopt p) (fun-formal p) (precompute-add (fun-body p)))]
    [(call? p) (call (precompute-add (call-funexp p)) (precompute-add (call-actual p)))]
    [(mlet? p) (mlet (mlet-var p) (precompute-add (mlet-e p)) (precompute-add (mlet-body p)))]
    [(apair? p) (apair (precompute-add (apair-e1 p)) (precompute-add (apair-e2 p)))]
    [#t #f]])

(define (complete-typechecker p) #t)

(define tpl-nats
  (mlet "f"
    (fun "f" "x"
      (apair (var "x") (fun null "z" (call (var "f") (add (var "x") (int 1))))))
    (fun null "unused" (call (var "f") (int 0))))
)
5. (12 points)

You are developing a kitchen simulation program in Ruby with classes to model various different kinds of physical containers: Jars, Bottles, Cups, Mugs, Glasses, Cannisters, Bags, Boxes, Cans, Trays, Cartons, Bowls, and more. In development, you’ve implemented quite the complicated system, which is perhaps too complicated to really be useful. The last time you ran the program, you ended up with boxes full of hot coffee, and a bunch of granulated sugar in bottles. None of this makes sense.

In order to understand what your errant program is currently doing, you decided to extract some debug information from all of the different classes which implement your different containers.

Your code is structured such that there is one Container class, and each specific container (e.g. Mug) is a subclass of Container.

(a) Write a mixin Debug which provides one method: describe. This method should take no arguments, and return a string which is the name of the runtime class of the object which describe is called on, as well as the result of the contents method (which returns what, if anything, is contained in a container).

(b) Which class(es) should include the mixin Debug such that every container will be able to describe itself accurately?

(c) You manage to track down what you were doing with boxes and bottles: a simple typo on one line. However, you are now running into another issue with some of your containers, but this seems limited to those meant for drinking (e.g. Mug or Glass). Unfortunately, you don’t have a DrinkingContainer class anywhere. Without adding any classes or changing the existing class hierarchy in any way, describe your strategy for changing the definition of describe for drinking containers, so that it will print out the type and volume of liquid contained inside (accessible by the total_volume and percent_full methods. Make sure to mention how you will make sure your new describe will be called, instead of your old definition. Explain in at most 2 sentences.

Solution:

(a) module Debug
   def describe
      contents + self.class.to_s
   end
   end

(b) The Container class.

(c) Various answers will work here. The key is that they show they understand the lookup order of methods in Ruby: if above they’ve included Debug in the Container class, then defining a different mixin and including it in all of the drinking container classes will make their new mixin get called. They should lose style points if they’ve used is_a or instance_of.
For this question, recall the game Rock Paper Scissors! If you’ve never played it before, all you need to know is that paper beats (covers) rock, rock beats (smashes) scissors, and scissors beats (cuts) paper.

Implement the classic Rock Paper Scissors! game using double dispatch in Ruby. Each object of the Rock, Paper, and Scissors classes should provide a play method, which takes one argument, and returns the winner. In the case of a tie, either result may be returned.

Example:

```ruby
r = Rock.new
p = Paper.new
p.play(r) #result should be p
r.play(p) #result should be p
```

Your code below:

```ruby
class Rock

end

class Scissors

end

class Paper

end
```
(continued) Now, implement the same functionality in SML. Your code should be one pattern match only, and you should use the minimum number of cases possible.

```sml
datatype move = Rock | Paper | Scissors

(* play : move -> move -> move *)
fun play left_move right_move =
```

Now, suppose that you want to extend your game to the classic Rock Paper Scissors Lizard Spock! Each move wins against 2 others, and loses against 2 others, but the probability that you and your opponent pick the same play is diminished. For reference, Scissors cuts Paper covers Rock crushes Lizard poisons Spock vaporizes Rock decapitates Lizard eats Paper disproves Spock. Would you rather extend your double dispatch implementation in Ruby, or your functional implementation in SML? Pick one, and explain why.

**Solution:**

```ruby
class Rock
  def play other
    other.playAgainstRock self
  end
  def playAgainstRock rock
    self
  end
  def playAgainstPaper paper
    paper
  end
  def playAgainstScissors scissors
    self
  end
end

class Paper
  def play other
    other.playAgainstPaper self
  end
  def playAgainstRock rock
    self
  end
end
```
def playAgainstPaper paper
    self
end

def playAgainstScissors scissors
    scissors
end
end

class Scissors
    def play other
        other.playAgainstScissors self
    end
    def playAgainstRock rock
        rock
    end
    def playAgainstPaper paper
        self
    end
    def playAgainstScissors scissors
        self
    end
end

datatype move = Rock | Paper | Scissors

fun play left_move right_move =
    case (left_move, right_move) of
    (Rock, Scissors) => Rock
    | (Paper, Rock) => Paper
    | (Scissors, Paper) => Scissors
    | (_, x) => x

I would answer SML, as the game can be implemented in only 11 cases in SML, whereas the Ruby double dispatch implementation would take 25 methods. If the student defended their answer well, they should get credit.
7. (Subtyping) (16 points) Consider a language similar to the language we saw in lecture, with (1) records with mutable fields, (2) higher order functions, and (3) subtyping.

Indicate whether the following subtypings are sound. No need for an explanation. We use $A <: B$ to indicate that $A$ is a subtype of $B$.

(a) $\{f_1 : \text{int}\} <: \{f_1 : \text{int}\}$
(b) $\{f_1 : \text{int}, f_3 : \text{string}\} <: \{f_3 : \text{string}, f_2 : \text{bool}, f_1 : \text{int}\}$
(c) $\{f_1 : \{x : \text{int}, y : \text{int}\}, f_2 : \{x : \text{int}\} \} <: \{f_2 : \{x : \text{int}, y : \text{int}\}, f_1 : \{x : \text{int}, z : \text{int}\}\}$
(d) $\{\} <: \{x : \text{int}, y : \text{int}, z : \text{int}\}$
(e) $\{x : \text{int}, y : \text{int}\} \to \{x : \text{int}, y : \text{int}\} <: \{x : \text{int}, y : \text{int}\} \to \{x : \text{int}\}$
(f) $\{x : \text{int}, y : \text{int}\} \to \{x : \text{int}, y : \text{int}\} <: \{x : \text{int}\} \to \{x : \text{int}, y : \text{int}\}$
(g) $\{a : \{x : \text{int}\} \to \{r : \text{int}\}\} <: \{a : \{x : \text{int}, y : \text{int}\} \to \{r : \text{int}\}\}$
(h) $\{x : \text{int}\} \to \{y : \text{int}\} \to \{x : \text{int}, y : \text{int}, z : \text{int}\} <: \{x : \text{int}, y : \text{int}\} \to \{x : \text{int}, y : \text{int}\} \to \{x : \text{int}, y : \text{int}\}$

Solution:

(a) Sound
(b) Unsound
(c) Unsound
(d) Unsound
(e) Sound
(f) Unsound
(g) Unsound
(h) Sound