CS 252, Takehome Exam 2 Name: _____

This is a 24 hour, open notes, books, etc. exam.
ASK if anything is not clear.
WORK INDIVIDUALLY.

If there are any corrections to exam questions, they will be posted to Canvas.

Question	Points	Score
1	15	
2	15	
3	15	
4	15	
5	15	
6	15	
7	10	
8	0	
Total:	100	

1. (15 points) Consider the following language.

$e ::=$		*Expressions*
	v	Value
	`inc` e	Increment operator
	`dec` e	Decrement operator
	`if` e `then` e `else` e	Conditional expressions
$v ::=$		*Values*
	i	Integers
	`true`	Boolean true
	`false`	Boolean false
$T ::=$		*Types*
	Int	Integer
	$Bool$	Boolean

The small-step operational semantics relation $e \to e'$ repeatedly evaluates an expression to another expression until the expression is a value and cannot be evaluated further.

$$[\text{IF-CTXT}] \quad \frac{e_1 \to e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \to \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3}$$

$$[\text{IF-TRUE}] \quad \frac{}{\texttt{if true then } e_1 \texttt{ else } e_2 \to e_1}$$

$$[\text{IF-FALSE}] \quad \frac{}{\texttt{if false then } e_1 \texttt{ else } e_2 \to e_2}$$

$$[\text{INC-CTX}] \quad \frac{e \to e'}{\texttt{inc } e \to \texttt{inc } e'}$$

$$[\text{INC}] \quad \frac{}{\texttt{inc } i \to i + 1}$$

$$[\text{DEC-CTX}] \quad \frac{e \to e'}{\texttt{dec } e \to \texttt{dec } e'}$$

$$[\text{DEC}] \quad \frac{}{\texttt{dec } i \to i - 1}$$

(Question continues on the next page.)

The following type system for this language contains multiple errors.

$$[\text{T-INT}] \quad \frac{}{i : Int}$$

$$[\text{T-TRUE}] \quad \frac{}{\texttt{true} : Bool}$$

$$[\text{T-FALSE}] \quad \frac{}{\texttt{false} : Bool}$$

$$[\text{T-IF}] \quad \frac{e_1 : T_1 \quad e_2 : T_2 \quad e_3 : T_2}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : T_2}$$

$$[\text{T-INC}] \quad \frac{}{\texttt{inc } e : Int}$$

$$[\text{T-DEC}] \quad \frac{}{\texttt{dec } e : Int}$$

(a) Write a sample program that highlights at least one error in the type system.

(b) Modify the typing rules to correct all errors in the typing system. **Be clear about which rules you are modifying.**

(c) Now add additional evaluation rules that would make the original, unmodified typing rules correct.

Submit a PDF with your answers to these questions.

2. (15 points) For this problem, your job is to create an `addMixin` function. It will add functionality to an object, similarly to how mixins work in Ruby.

   The `mixin.js` file is included with the exam material. The function skeleton is at the top of `mixin.js`. It takes in a target object (o) and another object with properties to be mixed in (mixin).

   You should return a Proxy object from this function. Override the `get` trap so that:

   - If the object already has a property, the object's property is returned.
   - If the object does not have the property, it returns the property from the mixin object.
   - If the property is `__original`, it returns the original object `o`. (This design provides a way to "unmix" a mixin.)
   - If none of the other cases hold, `undefined` should be returned as the result.

   You should not make any changes to this file outside of the `addMixin` function definition.

   The expected output of the program is given in the file `output_expected.txt`.

3. (15 points) This question implements a text-based display area. Set coordinates in the display are marked with "X". Unset coordinates are marked with ".".

   Implement the `update` and `run` methods in `graph.js` from the exam material. Details of how to implement these methods are given by comments in the file.

   To test the update method, run `node graph_test1.js` and verify your results match `graph_test1.output` exactly. Likewise, run `node graph_test2.js` and verify that your results match `graph_test2.output`.

   Finally, run `node graph_test3.js` to test out the run method. Due to the random nature, you will get different results on each run. (This simulation is somewhat famous, so you might recognize it).

4. (15 points) This question covers blocks and `method_missing` in Ruby.

   The file `node.rb` contains a `Node` class, representing a node on an undirected graph. Each node contains a key and a value. The key is a symbol, and the value can be whatever the user desires.

   First, implement an `each` method. This method takes an optional `visited` list, used to avoid backtracking, and a block. The block of code should be called, passing in the node's key and value. After that, the `each` method should be called for every connected, unvisited node.

   Next, update an `update_each` method. It works similarly to update, except that only the value should be passed in to the block, and that the result of the block should replace the current value of the node.

   (Note that the `find` method in this class might offer useful tips for the previous two methods.)

   Finally, implement `method_missing`. An unrecognized method name will cause the node to call `find` to search the graph for a connected node with a key that matches the method name.

   The expected result is given in `output_expected.txt`.

5. (15 points) This question covers regular expressions and singleton classes in Ruby.

   In `html-renderer.rb`, you must implement two methods as `static` methods, to borrow Java terminology.

   First, implement the `encode` method. It takes a string, replaces '<', '>', and '&' with the appropriate HTML entities, and then returns the modified string. (Comments in the code provide the HTML entities if you do not already know them).

   Second, implement a `render` method to replace the place holders in the template string with the matching args. So '$1' should be replaced with `args[0]`, '$2' with `args[1]`, '$3' with `args[2]`, etc. (Note the $ placeholders start with 1, while the `args` start with 0).

   The expected results are given in `expected_output.txt`.

   Note that the generated output will produce a valid HTML file.

6. (15 points) This problem reviews the lambda calculus.

   In this class, we have seen the difference between lazy evaluation (as in Haskell) and eager evaluation (as in most other programming languages).

   Provide *two sets* of evaluation rules for the lambda calculus. This first should follow an eager evaluation strategy, and the second should instead use a lazy evaluation strategy. Name your rules so that it is clear which set of rules uses which evaluation strategy.

   You may use either big-step or small-step operational semantics for this problem, but use the same style for both sets of rules.

   For reference, the language of the lambda calculus is given below.

$e ::=$		*Expressions*
	$(\lambda x.e)$	Lambda function
	x	Variable
	$e\ e$	Funciton applicaiton
$v ::=$		*Values*
	$(\lambda x.e)$	Lambda function

7. (10 points) When submitting your exam, include a text file with the following text:

   > I have not worked with anyone else for these exam problems. I have not consulted with any outside parties. For any code that I have used from an external source, I have cited the original source within my code comments.

8. (0 points) (Extra credit question: 5 points)

   This problem tests your understanding of borrows in Rust.

   Add the following two functions:

   1. The `find_second_largest` function should take in an array of i32s and return the second largest element.
   2. The `zero_out_pos` function should take in an array and an index. The specified position of the array should be set to zero.

   Starter code is provided in `extraCred.rs`, but be forewarned that you must provide matching function calls to your function – the code will not compile as-is.

   The expected output is given in `expected_output.txt`.