Q1
0 Points

Please **carefully read** the instructions below:

Ground Rules

This exam is open-note, which means that you may refer to your own notes and class resources during the exam. You can also use `irb` and `utop` (or other programs). You may **not** work in collaboration with anyone else, regardless of whether they are a student in this class or not. If you need to ask a question about the exam, post a private question on Piazza.

Sections

Section	Points
PL Concepts	**[10 pts]**
Regular Expressions	**[8 pts]**
Small Code: Fill in the Blanks	**[30 pts]**
Ocaml: Debugging	**[9 pts]**
OCaml: Coding	**[16 pts]**

General Advice

You can complete answers in any order, and we recommend you look through all of the questions before first so you can gauge how long you should spend on each question. Refer to the counter in the top left corner to ensure you have completed all questions.

Submission

You have 75 minutes to complete this exam (see the timer in the upper right corner for remaining time). Once you begin, you can submit as many times as you want until your time is up. You can even leave this page and come back, and as long as the time hasn't expired, you'll be able to update your submission. This means that if you accidentally submit, refresh, or lose internet temporarily, you'll still be able to work on the test until the time is up. If you come back, click "Resubmit" in the bottom-right corner to resume.

Honor Pledge

Please copy the honor pledge below:

> I pledge on my honor that I have not given or received any unauthorized assistance on this examination.

Signature

By entering your name above, you agree that you have read and fully understand all instructions above.

Q2 PL concepts

10 Points

Q2.1 Ruby Types

2 Points

At what point in a program's life are Ruby values checked in order to ensure that their types are consistent with their use?

- ○ When the program is compiled
- ○ When the program is run/executed
- ○ Ruby types are never checked

Q2.2 Typing I

2 Points

Manifest typing is a fancy way of saying "static typing".

- ○ True
- ○ False

Q2.3 Typing II

2 Points

At what point in a program's life are OCaml variables checked in order to ensure that their types are consistent with their use?

- ○ When the program is compiled
- ○ When the program is run/executed
- ○ OCaml types are never checked

Q2.4 Typing

2 Points

What is the type of the following expression?

```
let [x] = [[1]] in 1::x
```

- ○ `int list list`
- ○ `int list`
- ○ `'a list list`
- ○ `'a list`

Q2.5 Tail Position
2 Points

Can the following code take advantage of tail call optimization (is the functional tail recursive)?

```
let rec mystery x y =
  if y then (match x with
    |[] -> true
    |h::t -> mystery t (h && y))
  else false
```

- ○ Yes
- ○ No
- ○ Not enough information

Q3 Regex
8 Points

Q3.1 Counter Example
2 Points

Consider the following regex:

```
\d{2}:\d{2} (AM|PM)
```

This regex will match on valid 24-hour HH:MM times such as 12:38 PM and 02:00 AM. However, it will also match on inputs that do not represent valid 24 hour times. Provide an example of an input that the regex will accept, but is not a valid 24 hour time.

Q3.2
2 Points

Write the following regex without using `+`:

```
[a-zA-Z]+(\d*)
```

Q3.3
4 Points

Write a regex that matches on a comma-delimited list of integers and captures the first and last integer. You may assume the list is non-empty.

Valid Matches:

- 1
- 2,3,4
- 0,1,-1,2,-2

Invalid Matches:

- 1,
- 2.3,1.4,6.7
- a,b,c,d

Q4 Small Code
30 Points

Q4.1 Ruby Classes
6 Points

Define a class named `Population` with a class variable named `count` with an initial value of `0`. Define an instance method `increment` which increases `count` by `1`.

Q4.2 Property Based Testing
6 Points

Consider the following type:

```
type llist = Tail|Node of int * llist
```

Consider a function `rotate` that takes a `llist` and an `int` and rotates the list n times.

```
rotate Node(3,Node(2,Node(1,Tail))) 1 = Node(2,Node(1,Node(3,Tail)))
rotate Node(0,Node(1,Node(2,Tail))) 2 = Node(2,Node(0,Node(1,Tail)))
rotate Tail 1 = Tail
rotate Node(1,Tail) 2 = Node(1,Tail)
```

Suppose the following functions exist:

- `size lst: llist -> int` which takes in a `llist` and returns the length of the list
  - `size Tail = 0`
  - `size Node(1,Node(2,Node(3,Tail))) = 3`
- `mem x lst: int -> llist -> bool` which takes in an `int` and a `llist` and returns true if `x` is in the list
  - `mem 2 Node(1,Node(2,Node(3,Tail))) = true`
  - `mem 2 Tail = false`

Create 2 properties of `rotate` which use these functions (1 each)

Property using `size`

Description of property:

Code for property:

Property using `mem`

Description of property:

Code for property:

Q4.3
6 Points

Consider the following code in OCaml

```
let f = fun x -> fun y -> x + y;;
let g = f 3;;
let h = g 4;;
```

Write the above 3 lines in ruby to achieve the same functionality

```
f =
```

```
g =
```

```
h =
```

Q4.4 OCaml to a spec I
2 Points

Write an OCaml function, named `f` that has the following type:
`'a -> ('a -> 'b) -> 'b`

Q4.5 OCaml to a spec I

2 Points

Write an OCaml function, named `f` that has the following type:
`int -> (float -> string)`

Q4.6 Multiple Arguments

8 Points

In many languages, functions that take multiple arguments take all of their arguments 'at once'. In OCaml there are two ways for a function to take multiple arguments: 'at once' (using a tuple), just like in other languages, or 'one at a time'.

For example:

```
let at_once (x,y) =
    let _ = print_int x in
    print_int (x + y)

let one_at_a_time x y =
    let _ = print_int x in
    print_int (x + y)
```

In class we said that in OCaml, these are actually equivalent. You can 'prove' their equivalence by writing functions that convert between these two approaches. Implement the following two functions ensuring that your implementation matches the types below:

`val expand : ('a * 'b -> 'c) -> 'a -> 'b -> 'c`

`val contract : ('a -> 'b -> 'c) -> 'a * 'b -> 'c`

Q5 OCaml Debugging

9 Points

Consider the following code that takes a 3-tree and swaps the left and right of each sub-tree.

```
type tree = Leaf|Node of tree * tree * tree;;

let rec swap t = match t with
|Leaf -> Node(Leaf, Leaf, Leaf)
|Node(left, center, right) -> Node(swap left, center, swap right)
```

There are at least 3 bugs. Identify 3 unique bugs and state why it's a bug **and** the code to fix it.

Bug 1

Describe the bug

Code to fix the bug

Bug 2

Describe the bug

Code to fix the bug

Bug 3

Describe the bug

Code to fix the bug

Q6 OCaml Coding

16 Points

For the following questions, you may use the following definitions

```
let rec fold f a l = match l with
```

```
[]-> a
|h::t -> fold f (f a h) t;;

let rec foldr f l a = match l with
[] -> a
|h::t -> f h (foldr f t a);;

let rec map f l = match l with
[] -> []
|h::t -> (f h)::(map f t);;
```

Q6.1 OCaml Iteration

8 Points

Given an `'a list`, `xs`, and an `int`, `k`, write a function, `get_every`, that returns a list with every `k`th element of `xs`.

For our purposes, the ordering of the resulting list does not matter. In other words, it's okay if your approach does not result in the same lists as our examples below, *as long as your resulting list has the same elements*.

Hint: Use fold, and let your accumulator be of type int * ‘a list. Then use pattern matching to extract the list from the accumulator.

Examples:

```
get_every_kth 2 [1;2;3;4;5] = [2;4]
get_every_kth 3 [1;2;3;4;5;6;7;8;9;10] = [3;6;9]
get_every_kth 1 [“a”;”b”;”c”] = [“a”;”b”;”c”]
```

Q6.2 Graphs

Consider the following types

```
type node = Node of int;;
type graph = (node * node list) list;;
```

A `graph` is a list of nodes and a list of nodes each node points to.
Here is an example graph:

```
let g = [(Node(1),[Node(2);Node(3)]);
         (Node(2),[Node(3)]);
         (Node(3),[Node(1);Node(3)])];;
```

`g` has 3 nodes, `1`, `2`, and `3`.

- Node `1` points to Nodes `2` and `3`
- Node `2` points to Node `3`
- Node `3` points to Nodes `1` and `3`

Write an function `neighbor_value` which takes in a graph and returns a list of `int * int` tuples which represent a Node's value and the sum of it's neighbors. You may assume that each node has at least 1 neighbor.

`neighbor_value g = [(1,5);(2,3);(3,4)]`

You may **not** use the rec keyword but you **can** create helper functions.

```
let neighbor_value g =
```

Exam 1

● **UNGRADED**

14 DAYS, 7 HOURS LATE

STUDENT

Unknown Student (removed from roster?)

TOTAL POINTS

- / 73 pts

QUESTION 1

(no title)		0 pts

QUESTION 2

PL concepts		10 pts
2.1	Ruby Types	2 pts
2.2	Typing I	2 pts
2.3	Typing II	2 pts
2.4	Typing	2 pts
2.5	Tail Position	2 pts

QUESTION 3

Regex		8 pts
3.1	Counter Example	2 pts
3.2	(no title)	2 pts
3.3	(no title)	4 pts

QUESTION 4

Small Code		30 pts
4.1	Ruby Classes	6 pts

4.2	Property Based Testing	6 pts
4.3	(no title)	6 pts
4.4	OCaml to a spec I	2 pts
4.5	OCaml to a spec I	2 pts
4.6	Multiple Arguments	8 pts

QUESTION 5

OCaml Debugging 9 pts

QUESTION 6

OCaml Coding 16 pts

6.1	OCaml Iteration	8 pts
6.2	Graphs	8 pts