CMSC330 Fall 2016 Midterm #1
2:00pm/3:30pm

Name: ________________________________

Discussion Time:	**10am**	**11am**	**12pm**	**1pm**	**2pm**	**3pm**
TA Name (Circle):	**Alex**	**Austin**	**Ayman**	**Brian**	**Damien**	**Daniel K.**
	Daniel P.	**Greg**	**Tammy**	**Tim**	**Vitung**	**Will K.**

Instructions

- Do not start this test until you are told to do so!
- You have 75 minutes to take this midterm.
- This exam has a total of 100 points, so allocate 45 seconds for each point.
- This is a closed book exam. No notes or other aids are allowed.
- Answer essay questions concisely in 2-3 sentences. Longer answers are not needed.
- For partial credit, show all of your work and clearly indicate your answers.
- Write neatly. Credit cannot be given for illegible answers.

	Problem	Score
1	Programming Language Concepts	/13
2	Regular Expressions	/10
3	Ruby execution	/10
4	Ruby Programming	/15
5	OCaml Typing	/16
6	OCaml Execution	/18
7	OCaml Programming	/18
	Total	/100

1. Programming Language Concepts (13 pts)

A. (4pts) What will be the value of y in the following block of OCaml code, first evaluated with static scoping, and then re-evaluated with dynamic scoping:

```
let x = 10;;
let f =
    let square y = x * x in
       let x = 20 in
    square x;;
let y = f;;
```

Static:

Dynamic:

Solution:
Static: 100
Dynamic: 400

B. (2pts) Name an advantage and a disadvantage of using an interpreted language: (One sentence answer for each of advantage and disadvantage)

Many options here, e.g.

- **Fast to write/iterate on code**
- **code runs more slowly than compiled languages**
- **Code may be more portable (don't need to recompile)**

C. (1pt) One of the most important features of functional programming languages is
   a) Immutable data structures
   b) Statements with side effects
   c) Iterative control structures
   d) Implicit type definition

D. (1pt) Which is a **true** statement about lists in Ruby and OCaml?
   a) **Ruby lists are *heterogenous*; OCaml lists are *homogenous*.**
   b) Ruby lists are *homogenous*; OCaml lists are *heterogenous*.
   c) Ruby lists and OCaml lists are *heterogenous*.
   d) Ruby lists and OCaml lists are *homogenous*.

E. (1pt) Which of the following OCaml features is **only possible** because of closures:
   a) Tail Recursion
   b) Type Inference
   c) Partial Application
   d) Pattern Matching

F. (1pt) A typing system which allows the type of a variable to change within a function is considered to have _**Dynamic**_________ typing.

G. (1pt) A language which only lets the programmer use a variable after they have declared it is considered to have _**Explicit**____ declaration

H. (1pt) The `==` operator in Ruby and the `=` operator in OCaml are equivalent
   a) True
   b) False

I. (1pt) The terms “closure” and “function pointer” have the same meaning
   a) True
   b) False

2. Regular Expressions (10 pts)

A. (3 pts) Write a Ruby regular expression for a password that must have at least **1** letter and at least **1** digit.

\w*\d\w* | \w*[a-zA-Z]\w*

B. (2 pts) Give an example that adheres to the following regex:

```
/^[a-zA-Z0-9_\-.]+@[a-zA-Z0-9_\-.]+\.[a-zA-Z]{2,5}$/
```

Valid email address

C. (2 pts) Write the output of the following code:

```
"HIST100 (about the past)" =~ /[A-Z]{4}(\d{3}) (.+)/
puts $1
puts $2
```

100
(about the past)

D. (3 pts) Write the output of the following code: (Recall that `foo.inspect` gives the representation of `foo` as it would appear in source code, e.g. `[1,2,3].inspect` is `"[1,2,3]"`.)

```
s = "Computers need electricity."
a = s.scan(/[a-z]+/)
puts a.inspect
```

omputers
need
electricity

3. Ruby Execution (10 pts)

Write the output of the following Ruby programs. If the program does not execute due to an error, write **NO OUTPUT** instead. Recall that `foo.inspect` gives the representation of `foo` as it would appear in source code, e.g. `[1,2,3].inspect` is `"[1,2,3]"`.

A. (4 pts)

```
def myFun(x)
   yield x
end

myFun(3) { |v|
   str = v % 2 ? "foo" : "bar"
   puts "#{v} #{str}"
}
```

Output: **`3 foo`**

B. (3 pts)

```
x = [5, 10]
x[3] = 20
y = x
y << ["a", "b"]
puts x.inspect
```

Output: **`[5, 10, nil, 20, ["a", "b"]]`**

C. (3 pts)

```
foo = {1 => ["apple"], 2 => "kiwi", 3 => "yam"}
bar = foo.keys.sort { |a,b|
   foo[b].length <=> foo[a].length
}
puts bar.inspect
```

Output: **`[2, 3, 1]`**

4. Ruby Programming (15 pts)

You will be implementing a Ruby class named WordCounter which will read English text from a file and allow you to query the number of times a given word appears. For the purposes of this question, a word is a sequence of alphabetic characters. In our input files, words will be separated by whitespace and the following other characters: . , " : ;

You must implement the following methods. You may of course implement any other helper methods you wish. (5 points each):

initialize(filename): Reads the text from the file, extracts the words from the text, and stores them in an appropriate data structure. You may find it useful to use the IO.foreach method, which takes a filename and a code block, and passes each line in the file to the code block in turn, automatically closing the file when it reaches the end. Example usage:

```
IO.foreach("myfile.txt") { |line| puts line }
```

count(word): Returns the number of occurrences of word in the file. For the purposes of this question, words which differ by capitalization are considered different words, e.g. "Cat" and "cat" would be considered separate words.

each: Takes a code block and passes each word-and-count pair to it in turn. The order in which word-and-count pairs are passed to the block does not matter. You may assume that a code block will always be passed.

Example:

had.txt:

James had had "had," but John had had "had had."

example usage:

```
wc = WordCounter.new("had.txt")
puts wc.count("had")
wc.each do |word, count|
  puts "#{word}: #{count}"
end
```

output:

```
7
James: 1
had: 7
but: 1
John: 1
```

Solution:

```
class WordCounter
  def initialize(filename)
    @word_counts = Hash.new(0)

    IO.foreach(filename) do |line|
      words = line.split(/[\s.,":;]+/)
      words.each do |word|
        if word =~ /^[a-zA-Z]+$/
          @word_counts[word] += 1
        end
      end
    end
  end

  def count(word)
    @word_counts[word]
  end

  def each
    @word_counts.each do |word, count|
      yield(word, count)
    end
  end
end
```

Rubric and partial credit options

Note: In the past students have implemented assignments like this with varying amounts of processing in the constructor. We have given 2 free points for the constructor assuming that most students won’t check all of the validity constraints in the constructor, but instead in count and each. However, if the student checks all validity constraints in the constructor, the points for those constraints should be assessed on initialize and the free points given on count and each, to avoid double jeopardy.

5. OCaml Typing (16 pts)

Write Ocaml expression or definition of the following types **without using type annotation**:

```
A. (2 pts ) (int * float) list
```

```
B. (3 pts)  int -> string -> (int * string) list
```

```
C. (3 pts) 'a -> ('a -> bool) -> string
```

2. What is the type of the following expression?

```
A. (2 pts)     ([1;2], “foo”)
```

```
B. (3pts)   let f x y = x::(y x 2)
```

```
C. (3 pts) let rec f x = match x with
   |[] -> []
   |h::t -> match h with
           |(a,b) -> if (a = b) then a::(f t) else b::(f t);;
```

Solution:

1. Possible solutions:

```
1. [(3, 3.14)]
2. fun x y -> (x, y)::[(1, “hello”)]
3. fun x f2 -> if (f2 x) then “hello” else “hello world”
```

2.

```
1. int list * string
2. 'a -> (int -> 'a -> 'a list) -> 'a list
3. ('a * 'a) list -> 'a list
```

6. OCaml Execution (18 pts)

To the right of each code snippet, write what the variable `res` contains after executing the given code. Note that each code snippet contains syntactically and semantically valid OCaml code.

A. (3 pts)

```
let rec f x y = if y = 0 then x else f y (x mod y);;
let res = f 9 6;;
```

Output:3

B. (3 pts)

```
let a = ref "big";;
let b = ref "data";;
a := "buzz";;

let res = ((!a), (!b))
```

Output: (“buzz”, “data”)

C. (4 pts)

```
let proc f x y = if f x y > 0 then x else y;;
let res = proc (fun a b -> a * b) (-2) 4
```

Output:4

D. (4 pts)

```
let rec map f = function
 | []      -> []
 | x :: xs -> let z = f x in z :: map f xs
;;

let res =
 let f x y = x := !x + (y * y); !x in
 map (f (ref 0)) [1;2;3;4];;
```

Output: [1;2;6;24]

4 pts

E. **(4 pts)** (Hint: pervasive `max` of type `val max : 'a -> 'a -> 'a` returns the greater of the two arguments.)

```
type int_tree =
 | Leaf
 | Node of int * int_tree * int_tree
;;
let rec bar = function
 | Leaf           -> -1
 | Node(x, l, r) -> 1 + max (bar l) (bar r)
;;
let res =
 let t = (Node(1, Node(2, Leaf, Node(3, Leaf, Leaf)), Leaf)) in
 bar t
```

Output:2
4 pts

7. OCaml programming (18 pts)

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

let rec fold f a l = match l with
    [] -> a
   |h::t -> fold f (f a h) t
;;
Helper functions are allowed!
```

A. (6 pts) Using fold and/or map, write a function `multi_map` of type `('a -> 'a) -> ('a * int) list -> 'a list`. This new higher order function applies the input function to each element of the input list however many times are specified by the second element of the tuple. Output order must be the same as the input order. If a negative int is encountered, do not apply the function to that element.

Example input:
multi_map (fun x -> x * x) [(5, 1); (7, -1); (2, 3); (10, 2)]
> [25, 7, 256, 10000]

Solution:

```
let rec multi_map f l =
        let rec apply e n = if n <= 0 then e else apply (f e) (n - 1) in
        match l with
        | [] -> []
        | (e, n)::t -> (apply e n)::(multi_map f t)
```

B. (6 pts) Using fold and/or map, write a function `relative` of type `int list -> int list`. This function decreases each element in the input list by the list's smallest integer.

Example input:

```
Relative [100; 80; 90]
> [20; 0; 10]
```

Solution:

```
let relative l = match l with
   | [] -> []
   | h :: t ->
      let s = fold min h t in
      map (fun h -> h - s) l
```

C. (6 pts) Using fold and/or map, write a function `max_repeat` of type `‘a list -> int`. This function finds the maximum number of times an element of the input list is repeated in a row. If the list is empty, the result should be 0.

Hint: The pervasive `max` might make your implementation easier.

```
val max : 'a -> 'a -> 'a
```

Return the greater of the two arguments.

Example input:

```
max_repeat [1; 2; 2; 2; 0; 3; 3; 3; 3]
> 4
```

Solution:

```
let max_repeat = function
   | [] -> 0
   | h :: t ->
       let aux (c, n, m) h =
           if c = h then (c, n + 1, m)
           else (h, 1, max m n)
       in
       let (_, n, m) = fold aux (h, 1, 1) t in
       max n m
```