CMSC 330, Fall 2018 — Midterm 1

Name _____

Teaching Assistant

Kameron Aaron Danny Chris Michael P. Justin Cameron B. Derek Kyle Hasan Shriraj Cameron M. Alex Michael S. Pei-Jo

Instructions

- Do not start this exam until you are told to do so.
- You have 75 minutes for this exam.
- This is a closed book exam. No notes or other aids are allowed.
- For partial credit, show all your work and clearly indicate your answers.

Honor Pledge

Please copy and sign the honor pledge: "I pledge on my honor that I have not given or received any unauthorized assistance on this examination."

_____

_____

_____

_____

Section	Points
Programming Language Concepts	10
Ruby Regular Expressions	10
Ruby Execution	12
Ruby Programming	18
OCaml Typing	17
OCaml Execution	15
OCaml Programming	18
Total	100

Programming Language Concepts

1. [1 pts] (T / F) In every programming language, code must be compiled before it is run.
2. [1 pts] (T / F) Static typing occurs during program execution and dynamic typing occurs before the program is run.

In the following questions, circle **all** answers that apply.

3. [2 pts] In Ruby, which of the following are objects?

   (a) `true` (b) `Hash.new` (c) `2` (d) `[1]`

4. [2 pts] In OCaml, which of the following are true about functions?

   (a) They can take other functions as arguments.

   (b) They have to be given a name to be used.

   (c) They will throw an error if not given enough arguments.

   (d) They can return another function as an output.

5. [2 pts] Which of the following is stored in a closure?

   (a) the execution stack (b) the function's output (c) the environment (d) the function's code

6. [2 pts] Which of the following fit the functional programming paradigm?

   (a) loops

   (b) recursion

   (c) higher-order functions

   (d) mutable variables

Ruby Regular Expressions

1. [3 pts] Give the output of the following code snippet.

```
secret = "<a href='umd.edu' user='cmsc330' />"
result = secret.scan(/([a-z]+)(=)/)
puts(result)
```

2. [2 pts] Given the regular expression `/Oh. What's that?b*/` circle all of the strings that are matched entirely. (Hint: `?` means zero or one.)

   (a) Oh. What's thatb

   (b) Oh. What's that?

   (c) Oh! What's that

   (d) Oh. What's thabb

3. [2 pts] What is the output of the following?

```
"href=https://www.ign.com/ps4" =~ /.{3}\/([a-z]+\d)/
puts($1)
```

4. [3 pts] Write a Ruby regular expression to match room numbers. Room numbers are a building code (three capital letters), followed by a space, followed by a room code (four digits, possibly with a B in front). Examples are given below.

   ESJ 0202 CSI 1115 MTH B0421

Ruby Execution

Next to each Ruby snippet, write the output after executing it. If there is an error, then write "error."

1. [2 pts]

```
x = {"CMSC330" => 1, "CMSC351" => 2, "CMSC320" => 3}

x["CMSC216"] = 4

if x.key?("CMSC216") then
  x["CMSC330"] = x["CMSC216"]
end

puts(x["CMSC330"])
```

2. [2 pts]

```
i = 0
x = Hash.new(1)
3.times do
   x[i] += 1
   i += 1
end
puts(x)
```

3. [2 pts]

```
x = [1, 2, 3]
x << "four"

puts(x)
```

4. [3 pts]

```
def fun(y)
  i = 1
  while i <= y
    yield(i * i)
    i += 1
  end
end

fun(4) { |x|
  puts(x)
}
```

5. [3 pts]

```
class Foo
  @@y = []
  def initialize(ele)
    @@y.push(ele)
  end

  def add(ele)
    @@y.push(ele)
    @@y
  end
end

f = Foo.new(12)
g = Foo.new("a")
h = g.add("e")
h[3] = "i"

puts(g.add("o"))
```

Ruby Programming

You're in charge of building a Ruby class for a bank that deals exclusively in the tech world's hottest new currency: UMDCash. You need to implement the `Bank` class in order to support this hot new economy and make the teaching staff very rich. The `Bank` class must store the account names and each account's current tally of UMDCash. You can assume that (a) each account name is distinct, and (b) there are no negative amounts ever provided to your code.

1. [2 pts] Implement `initialize`. You will have to decide the proper data structure(s) for storing the data needed for the rest of the outlined functions.

2. [8 pts] Implement `importAccounts` that takes the name of a file and loads accounts and their existing UMDCash tally. The file has one account per line, in the format "name:amount" where the name consists of (upper and lowercase) letters and the amount is an integer or float. If the same account appears more than once in the file, store only the highest value. If any line doesn't follow this format, the line is invalid and should be skipped. Multiple calls to `importAccounts` will update existing balances. Names are NOT case sensitive (i.e. boB and Bob will be treated as the same name).

   Here is an example.

   ```
   b = Bank.new
   b.importAccounts("file.txt")
   ```

   Here is the contents of `file.txt`.

   ```
   Bob:10
   Alice:48.50
   McKenzie:33
   Samantha:28.01
   Kyle:Fifty # Invalid Line
   ```

   You may find it useful to use the `IO.foreach` method, that reads files.

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

3. [4 pts] Implement `transfer` that takes in an account name `from`, account name `to`, and a transfer amount. It returns `true` if the transaction succeeds, or `false` if either of the accounts do not exist or the `from` account doesn't have enough UMDCash to complete the transfer. If `false` is returned, neither account's balance should change.

4. [4 pts] Implement `whoIsTheRichest` that returns the name of the account in your bank who currently has the highest UMDCash total. Return `nil` if there are currently no accounts in your bank.

Implement your solution in the class skeleton below.

```
class Bank
  def initialize()



  end

  def importAccounts(file)

  end

  def transfer(from, to, amount)

  end

  def whoIsTheRichest()

  end
end
```

OCaml Typing

Recall the definition of `'a option`.

```
type 'a option = Some of 'a | None
```

Determine the type of the following expressions. If there is an error, write "error."

1. [3 pts]

```
fun a -> if a then None else (Some a)
```

2. [3 pts]

```
fun a b c -> (a b) +. c > 0.0
```

Write an expression that has the following type, without using type annotations.

3. [3 pts] `int -> int option -> int`

4. [3 pts] `('a -> 'b -> 'a * 'b) -> 'a -> 'b -> ('a * 'b) list`

For each question, define a function `f` that when used in the following expressions will not produce any type errors.

5. [3 pts]

```
let f g t =
```

```
in fun g t -> [g t; f g t]
```

6. [2 pts]

```
let f x =
```

```
in fun x -> (f x) :: [x]
```

OCaml Execution

Recall the definitions of `map`, `fold_left`, and `fold_right`.

```
let rec map f xs =
  match xs with
  | [] -> []
  | x :: xt -> (f x) :: (map f xt)
```

```
let rec fold_left f a xs =
  match xs with
  | [] -> a
  | x :: xt -> fold_left f (f a x) xt
```

```
let rec fold_right f xs a =
  match xs with
  | [] -> a
  | x :: xt -> f x (fold_right f xt a)
```

Write the final value of the following OCaml expressions next to each snippet. If there is an error, write "error."

1. [2 pts]

```
let y = 3 in
let x = y + 1 in
let y = 5 in
x
```

2. [2 pts]

```
let f x y = if x > y then x - y else (if x < y then y - x) in
f 3 2
```

3. [3 pts]

```
let f (a1, a2) = a1 * a2 in
map f [(2, 2); (1, 4); (3, 2)]
```

4. [4 pts]

```
fold_left (fun a x -> if x mod 3 = 0 then x :: a else a) [] [1; 3; 11; 27]
```

5. [4 pts]

```
let rec f l = match l with
| [] -> []
| h :: t -> (fold_left (fun a x -> a * x) 1 t) :: (f t) in
f [1; 2; 3; 4]
```

OCaml Programming

For the following questions you may use `map`, `fold_left`, and `fold_right`.

1. [8 pts] Write `sum_thresh : int list -> int -> (int * int)` that returns a tuple where

   - the first component is the sum of all elements in `xs` strictly less than `thresh`, and
   - the second component is the sum of all the elements in `xs` greater than or equal to `thresh`.

   Examples:

   ```
   sum_thresh [5; 2; 7; 3] 4 = (5, 12)
   sum_thresh [1; 2; 10; 5] 5 = (3, 15)

   let sum_thresh xs thresh =
   ```

2. [10 pts] Write `compress: 'a list -> ('a * int) list` that packs in consecutive duplicate elements together in a tuple `(a, b)`, where `a` is the element and `b` is the number of consecutive occurrences.

   Examples:

   ```
   compress [] = []
   compress ["a"; "a"] = [("a", 2)]
   compress ["a"; "a"; "b"; "c"; "b"; "b"] = [("a", 2); ("b", 1); ("c", 1); ("b", 2)]

   let compress xs =
   ```