CMSC330 Spring 2015 Midterm #2
9:30am/11:00am/12:30pm

Name: ________________________________ **UID:** ____________________

Discussion Time (circle one): **10am 11am 12pm 1pm 2pm 3pm**
Discussion TA (circle one): Amelia Casey Chris Mike Elizabeth Eric Tommy

Instructions

- The exam has 3 pages; make sure you have them all.
- 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	OCaml typing	/19
2	OCaml execution	/12
3	OCaml coding	/20
4	Short answer, true/false	/18
5	Multithreading	/10
6	Lambda calculus	/12
7	Context Free Grammars	/9
	Total	/100

1. **OCaml typing (19 pts)**

   a. (3 pts) What is the type of the following OCaml expression?

```
[[1]; [2; 3]]
```

Answer: `int list list`

   b. (3 pts) What is the type of `f` in the following definition?

```
let f seed =
  let r = ref seed in
  (fun x -> r := !r + x; !r);;
```

Answer: `int -> int -> int`

   c. (4 pts) What is the type of `foo` in the following definition?

```
let rec foo (x,y,l) =
  match l with
    [] -> []
  | (h::t) ->
      if (h = x) then (y::(foo (x,y,t)))
      else (h::(foo (x,y,t)));;
```

Answer: `‘a * ‘a * ‘a list -> ‘a list`

   d. (3 pts) Write an OCaml expression having type `int -> int`

Answer (sample): `fun x -> x + 1`

   e. (3 pts) Write an OCaml expression having type `‘a -> ‘a list`

Answer (sample): `fun x -> [x]`

   f. (3 pts) Write an OCaml expression having type `bool * int -> int`

Answer (sample): `fun (x,y) -> if x then y else 1`

2. **OCaml execution (12 pts)**: What is the value of the variable `result` after executing the following code? If an error will occur, say what the problem is. (Note there are no syntax errors in these programs.)

   a. (3 pts)

```
let g x =
  match x with
    h::t -> h;;
let result = g [[9; 17]];;
```

Answer: `[9;17]`

   b. (3 pts)

```
let result = (fun x y -> x::y) (3 [4]);;
```

Answer: `ERROR, attempt to apply 3 as a function`

   c. (3 pts)

```
let rec foo (x,y,l) = match l with
   [] -> []
 | (h::t) ->
   if (h = x) then (y::(foo (x,y,t)))
   else (h::(foo (x,y,t)));;

let result = foo("y","p",foo("p","c",["p";"l";"a";"y"]));;
```

Answer: `["c";"l";"a";"p"]`

   d. (3 pts)

```
let f seed =
  let r = ref seed in
  (fun x -> r := !r + x; !r)
;;
let g = f 17;;

let result = (g 1, g 2, g 3);;
```

Answer: `(23,22,20)`

Partial credit (2 pts): (18,20,23)

3. **OCaml coding (20 pts). Complete two of the following three questions**. If you do all three, all three will be graded, and the total will be scaled by ⅔ (to be at most 20). You may write helper functions if you like. You may also use standard library functions, e.g., from the `List` and `Pervasives` modules, unless otherwise noted.

   a. (10 pts) Write a function `multiply`, having type `int -> int -> int`, that returns the product of its two arguments. You may assume that both arguments are nonnegative, and your implementation may use the built-in + and - operators, but not any other built-in arithmetic operators.

Answer:

```
let rec multiply x y =
  if x = 0 then 0
  else y + multiply (x-1) y
```

   b. (10 pts) Write a function get_val having type `‘a list -> int -> ‘a option` such that `get_val` *x* *n* returns (`Some` *z*) if *z* is the element of list *x* at index *n* (indexes start at 0) and returns `None` otherwise (i.e., if the list has at fewer than *n*+1 elements). Recall that

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

   Examples:

```
get_val [“hello”; “there”; “friend”] 0    ⇒    Some(“hello”)
get_val [1; 2; 3] 1                       ⇒    Some(2)
get_val [1] 2                             ⇒    None
```

Answer:

```
let rec get_val l i =
  if i<0 then None else
    match l with
      [] -> None
    | h::t -> if i = 0 then (Some h) else get_val t (i-1)
```

c. (10 pts) Given the type `g_tree` defined as follows:

```
type g_tree =
  GNode of g_tree list
| Leaf
```

Write a function `count` of type `g_tree -> int` that counts the number of leaves. It must take linear time (traverse the tree once). For example

```
count GNode([GNode([Leaf]);Leaf;Leaf])          ⇒    3
count GNode([])                                 ⇒    0
```

Answer:

```
let rec count t =
  match t with
    Leaf -> 1
  | GNode(l) ->
      let rec aux gs =
        (match gs with [] -> 0 | h::t -> (count h) + (aux t)) in
      aux l
```

or

```
let rec count t =
  match t with
    Leaf -> 1
  | GNode(l) ->
      List.fold_right (fun t a -> (count t) + a) l 0
```

4. **Short answer, true/false (18 points).**

   a. (3 pts) When using closures to encode objects, where are an object's private fields stored, so that they are private?
   Answer: In the closure's environment

   b. (3 pts) What is the relationship between a *wait set* and a *condition variable*?
   Answer: A wait set keeps track of the threads waiting on a condition variable

   c. (3 pts) True or false: In a type-safe language, all type-correct programs are well-defined.
   Answer: true

   d. (3 pts) True or false: *static type checking* occurs once the program starts to execute.
   Answer: false

   e. (3 pts) True or false: *reentrant* locks are important for avoiding inadvertent self-deadlock.
   Answer: true

   f. (3 pts) True or false: if OCaml had dynamic scoping, it would give the same answer as static scoping on the following program:

```
let f x = x+1;;
let g y = (f y) + 1;;
let f x = x-1;;
g 2;;
```

   Answer: false

5. **Multithreading (10 points).**
   a. (6 points) Given the following Ruby code, give two possible legal *schedules* in which the final value of `$var` is different; for each schedule also give the final value for `$var`. A schedule is a sequence of events indicating the perceived order in which threads execute particular lines of code. For example, the schedule T1:1, T1:2, T2:1, T2:2 says that thread T1 executes line 1, then T1 executes line 2, then T2 executes line 1, and finally T2 executes line 2.

```ruby
$var = 1

def driver_1()
  local_x = $var        # line 1
  $var = local_x + 1    # line 2
end

def driver_2()
  local_y = $var        # line 3
  $var = local_y + 2    # line 4
end

t1 = Thread.new{ driver_1() }
t2 = Thread.new{ driver_2() }
```

Answer(s):
T1:1, T1:2, T2:3, T2:4 $var=4
T1:1, T2:3, T2:4, T1:2 $var=2
T1:1, T2:3, T1:2, T2:4 $var=3
(there are others)

   b. (4 pts) Mark up the code above to show how you could use synchronization to ensure that only one final value of `$var` is possible.

Answer: Create a monitor, put contents of driver_1() and driver_2() in synchronize block.

6. **Lambda calculus (12 points).** Evaluate the following lambda terms as much as possible. For full credit, show each beta reduction and alpha conversion you perform.

a. (3 pts) `(λx.x) z`
Answer: `(λx.x) z → z`

b. (3 pts) `(λx.λy.x y) z y`
Answer: `(λx.λy.x y) z y → (λy.z y) y → z y`

c. (3 pts) `(λz.λy.z y) y x`
Answer: `(λz.λy.z y) y x = (λz.λw.z w) y x → (λw.y w) x → y x`

d. (3 pts) Recall the encoding of Church numerals:

```
1 = λf.λy.f y
2 = λf.λy.f(f y)
3 = λf.λy.f(f(f y))
(+ M N) = λx.λz.(M x)((N x) z)
```

The following applies beta reductions to `(+ 1 2)` and seems to end up with the term 2, instead of 3. There is a problem in one of the beta reductions (the grayed part); circle the problem and write a note to explain what's wrong.

`(+ 1 2)`	*//Substitute encoding for +*
`= `**`λx.λz.(`**`1`**` x)((`**`2`**` x) z)`**	*//Substitute encoding for 1*
`= λx.λz.(`**`(λf.λy.f y)`**` x)((2 x) z)`	*//Beta*
`→ λx.λz.(λy.x y)((2 x) z)`	*//Beta*
`→ `**`λx.λz.((2 x) z)`**	*//Substitute encoding for 2*

(BUG) in the above substitution: should have been λx.λz.(x ((2 x) z))

`= λx.λz.((`**`(λf.λy.f(f y))`**` x) z)`	*//Beta*
→ λx.λz.(((λy.x(x y)) z)	*//Beta*
→ λx.λz.(x(x z))	*//Alpha conversion*
`= λf.λy.(f(f y))`	*//Unsubstitute encoding for 2*
`= 2`	

Context free grammars (9 pts). Consider the following grammar (where uppercase letters are non-terminals, lowercase letters and symbols are terminals, and S is the start symbol).

S → a | & S | S ! T | T b
T → a | b | ε

e. Indicate whether the following strings are in the language accepted by this grammar. If a string is in the grammar, show parse justifying it. For example, the string &&ab is in the grammar, justified by the following parse:
S → &S → &&S → &&Tb → &&ab

i. (3 pts) &a&a!b

Answer: false (no way to get &a& pattern)

ii. (3 pts) ab!

Answer: S → S!T → Tb!T → ab!T → ab!

f. (3 pts) Is this grammar ambiguous? Why or why not?

Answer: Yes, because there are multiple ways to parse the string &a!b:

S → S!T → &S!T → &a!T → &a!b
S → &S → &S!T → &a!T → &a!b