CMSC330 Spring 2019 Final Exam

Name (PRINT YOUR NAME as it appears on gradescope):

____

Instructions

- The exam has 18 numbered pages; make sure you have them all.
- Do not start this test until you are told to do so!
- You have 120 minutes to take this exam.
- This exam has a total of 130 points, so allocate 55 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	PL Concepts	/8
2	OCaml	/36
3	Ruby	/15
4	Rust	/15
5	Regexps, FAs, CFGs	/22
6	Parsing	/8
7	Security	/12
8	Lambda calculus, operational semantics	/14
	TOTAL	/130

1. PL Concepts [8 pts]

Circle the T for each statement that is true, and F for those that are false.

Different implementations of the same abstract data type (ADT) can safely interoperate	T / F
You can encode an object in a functional language using a record of closures	T / F
A type safe language is one in which well-typed programs cannot go wrong	T / F
When viewed as a uni-typed static system, a dynamic language's type system is, like most static type systems, sound but not complete	T / F
Finite automata can be used to define the syntax of valid Java arithmetic expressions, but not the syntax of valid Java statements	T / F
The Rust language does not prevent memory management errors such as use-after-free and double-free because it does not use garbage collection	T / F
The OCaml compiler can optimize tail recursive function calls to use less memory	T / F
Copying garbage collection is slower than mark-sweep collection because it requires tracing the entire heap, not just the parts that are still live	T / F

2. OCaml [36 pts]

A. [6 pts, 2 each] **Write the types** of the following OCaml definitions. Write “type error” if the code doesn’t type check, and briefly give a reason why.

1. `let f x y z = (x + y) :: z :: ['2']`

2.
```
let make () =
  let x = ref 0 in
  ((fun y -> let z = !x in x := !x + y; z),
   (fun () -> !x))
```

3.
```
let bind f x = match x with
  None -> None
| Some n -> Some (f n)
```

B. [6 pts, 2 each] Provide **expressions**, *without type annotations*, that have the following types.

1. `bool -> int -> int * int`

2. `'a list -> 'b list -> ('a * 'b) list`

3. `rectangle -> float`

   where

   `type rectangle = { width: float; length: float }`

C. [6 pts, 2 each] Write the **result of evaluating the following OCaml expressions**. If there is a type error, write “type error” and briefly explain why. (There are no syntax errors.) The implementations of `map` and `fold` are given here.

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

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

1. `map (fun x -> bind (fun y -> y + 1) x) [None; Some 2; Some 3; None]`
   where `bind` is defined in part A.3, above

2.
   ```
   map (fun (x,y) -> if x > y then 0.0 else 1.0)
        [(1,2);(4.0,3.0);(true,false)]
   ```

3.
   ```
   fold (fun (x,m) (y,n) -> if x=y then (y,n+1) else (x,m))
         (1,1)
         [(2,1);(3,1);(1,2)];;
   ```

D. [8 pts] **Implement the function** `multiplicity : 'a list -> ('a * int) list`. The output list of tuples (the *multiplicity list*) indicates how many times an element occurs in the input list. The order of elements in the output list is unimportant. Examples:

```
multiplicity [true;true;false;false;true] = [(true,3);(false,2)]
multiplicity [1;2;5;2;3;1;1]             = [(1,3);(3,1);(2,2);(5,1)]
```

You may implement `multiplicity` however you like, but here is a ***hint***: Implement a helper function `ins : ('a * int) list -> 'a -> ('a * int) list`, which takes a multiplicity list and an element, and incorporates that element into the list. Your `multiplicity` function repeatedly applies `ins` to the input list to produce the final result. You will get partial credit if you implement `ins` even if you don't get the whole answer.

E. [6 pts] **Implement the function** `max_score : 'a list -> ('a -> int) -> int`. This function takes a list and a scoring function. The `max_score` function returns the maximal score of the given list's elements, or throws exception `Failure "empty"` if the input list is empty. (Again, any OCaml language features or libraries may be used.)

Examples:

```
max_score [1;2;5] (fun x -> x)                  = 5
max_score [[1;2];[3;4];[6];[]] List.length      = 2
```

F. [4 pts] **Write a function** you should pass to `max_score` so that it returns the maximal sum of a given list of `int lists`. I.e., write a function `foo` (using any OCaml and libraries you like) so the following calls work as written.

```
max_score [[0;5;-3];[8;2;-12];[1;1;4]] foo      = 6
max_score [[1;3;4];[4;4;2]] foo                 = 10
```

3. Ruby [15 pts]

A. The Iribe Center is the hottest building on campus. Everyone wants to book their events there. The CS department has been asked to create an automated system that will track professors' events, specified in a datafile. Each line of the provided datafile has 4 parts: **room number** (can be either 3 or 4 digits long), **last name** (must start with a capital letter and all subsequent letters must be lower case), **date** (given in mm/dd/yyyy format), and **time** (given in hh:mm format). Each part is comma-separated, with one space after each comma and no space before or after the colons. Below are some examples:

```
room:1250, name:Mamat, date:05/20/2019, time:09:30
room:130, name:Hicks, date:11/03/2019, time:12:05
room:2025, name:Eastman, date:09/01/2020, time:15:45
room:1250, name:Hicks, date:09/01/2020, time:10:15
```

1. [6 pts] **`add_to_system(datafile):`** This function should add each event given in `datafile` to a hash named `reservations`, where the key is the room number of type `String` and the corresponding value is an array of arrays, where the latter have the form `[lastname, date, time]`, each element of which is of type `String`. In your code you may call a function `valid_day_time(date,time)` which takes a date as a `String` in mm/dd/yyyy format and a time as a `String` in hh:mm format and returns `true` if they are legal, and `false` if not. Only insert lines with a valid date and time into `reservations`. For example, `valid_day_time("50/20/1971","10:20")` and `valid_day_time("7/20/1970","25:20")` would both return `false`. Do not worry about double-booking a room due to overlapping events. For the example datafile above, **`add_to_system`** would return the following `reservations` hash

```
reservations = {    "1250" =>    [["Mamat", "05/20/2019", "09:30"],
                                   ["Hicks", "09/01/2020", "10:15"]],
                    "130" =>     [["Hicks", "11/03/2019", "12:05"]],
                    "2025" =>    [["Eastman", "09/01/2020", "15:45"]]  }
```

```
def add_to_system(datafile)
  reservations = {}
  IO.foreach(datafile) { |line|









  }
  return reservations
end
```

2. [4 pts] **`get_events(reservations,name)`**: Given the `reservations` hash returned by `add_to_system`, and a professor's name, return an array of that professor's events, where each event is an array `[room#, date, time]`. So, `get_events(reservations,"Hicks")` would return `[["1250","09/01/2020","10:15"], ["130","11/03/2019","12:05"]]` for our example.

```
  def get_events(reservations, name)
    events = []





    return events
  end
```

3. [5 pts] **`busiest_day(reservations)`**: This function should return the date (as a string) in the given `reservations` hash with the largest number of events. If multiple dates have the same maximal number of events, return any one of them. If the reservations hash is empty, return the string `"No Reservation Found"`. So, `busiest_day(reservations)` would return `"09/01/2020"` for our example.

```
  def busiest_day(reservations)





  end
```

4. Rust [15 pts]

A. Using the code snippet below, answer problem A.1 and A.2. You can assume the code compiles and that there are no ownership errors:

```
1     fn iribe(){
2         let x = vec![true, false, true];
3         let s1 = x;
4         let s2 = center(&s1);
5         let s3 = computing(s1);
6         println!("It’s {} not {}!", s2, s3);
7     }
8     fn computing(s4: Vec<bool>) -> bool {
9         s4[1..2].iter().fold(true, |a,b| a && *b)
10    }
11    fn center(s5: &Vec<bool>) -> bool {
12        s5.iter().fold(false, |a,b| a || *b)
13    }
```

1. [3 pts] What variables have ownership of the vector `vec![true, false, true]` at any point during the call to `iribe()`?

☐ **x** ☐ **s1** ☐**s2** ☐**s3** ☐**s4** ☐**s5**

2. [2 pts] *After* the execution of which line would the vector `vec![true, false, true]` dropped)?

____

B. [6 pts] Using the code snippet below, answer problem:

```
1     fn final(){
2         let s1 = 5;
3         for i in 0..5 {
4             s1 = s1 + 1;
5         }
6         let mut s2 = vec![4,5,6];
7         if s2[1] > 0 {
8             let s3 = 5;
9         }
10        let s4 = s2.get_mut(2);
11        *s4 += 1;
12        let s5 = s3 + s2.get(1).unwrap();
13        let mut s6 = vec![1,2,3];
14        let mut r = &s6;
15        r[0] += 1;
16        let st = Some(String::from("string"));
17        let rf = &st;
18        match rf {
19            Some(x) => println!("It's {}", (x+1)),
20            &None => println!("There's nothing.")
21        };
22    }
```

The above code has 5 mistakes in it. *There are no syntax errors*; all errors are either *type*, *lifetime*, or *borrow* errors. Each error is on its own individual line. **Find any 3 of the 5 errors** and identify what type of error is being made**. Write the line number of each error and select the error type**. No credit for any error beyond the first 3.

Line Number	Error Type		
___________	☐**Type Error**	☐**Lifetime Error**	☐**Borrow Error**
___________	☐**Type Error**	☐**Lifetime Error**	☐**Borrow Error**
___________	☐**Type Error**	☐**Lifetime Error**	☐**Borrow Error**

For reference, here are function signatures for some of the methods used above:

`Vec<T>: fn get(&self, index: usize) -> Option<&T>`

Returns the element of a slice at the given index, or None if the index is out of bounds.

`Option: fn unwrap(self) -> T`

Moves the value v out of the `Option<T>` if it is `Some(v)`.

`Vec<T>: fn get_mut(&self, index: usize) -> Option<&mut T>`

Returns a mutable reference to an element at `index`; `None` if `index` is out of bounds.

C. [4 pts] Suppose we want to write a program manages a roster of students, implemented as a vector. A student is defined by the following record:

```
struct Student {
   name:String,
   age:i32,
   uid:i64
}
```

Implement the function `add_student`, which adds a new student to the end of the given `roster` vector, returning nothing. Only students of age 21 and older who are not currently in the roster (based on their UID) should be added. Here is an example call:

```
add_student(&mut Vec::new(),
            Student {name: String::from("James"), age:18, uid:112345});
```

Write your code here:

```
fn add_student(roster:&mut Vec<Student>, student:Student)  {









}
```

Some additional, useful `Vector` function names/prototypes, for your reference:
`iter(), push(element:T), as_slice(), insert(index: usize, element:T), len(), pop(), is_empty(), remove(index:usize)`

5. Regex, FAs, and CFGs [22 pts]

A. [2 pts] Circle all of the strings that will be accepted by the regexp (a*b)|(ba).

ab   aabba   ababab   b

B. [4 pts] Draw an NFA for the same regular expression (a*b)|(ba)

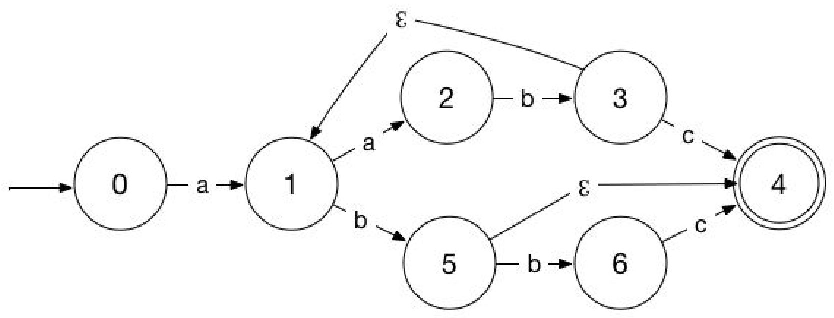
C.[4 pts] Given the following grammar, write a rightmost derivation of “(1+2)*3”, showing all steps of the derivation.

```
S -> S + S | A
A -> A * A | N
N -> 1 | 2 | 3 | (S)
```

D. [4 pts] Rewrite the grammar in part C above so that it is unambiguous and can be parsed by a recursive descent parser.

E. [3 pts] Write a regular expression with the alphabet Σ = {0,1} that exclusively accepts strings that begin and end with 1 and if 0s occur in between, there must be at least two, consecutively. Examples: 11, 1001, 11001, 1001001, 100011

F. [5 pts] Convert the following NFA into a DFA.



6. Parsing [8 pts]

A. [3 pts] For the following grammar, **write out the *first sets*** for each of the nonterminals

S -> iA | rBi
A -> eB | ε
B -> (S) | be | Ai

First(S) = {   }

First(A) = {   }

First(B) = {   }

B. [5 pts] The following code implements a recursive descent parser. **Write a CFG** that expresses the strings accepted by this parser.

Answer:

```
exception ParseError of string

type tok = Tok_a | Tok_t | Tok_e | Tok_d

let (tok_list : tok list ref) = ref [];;


let lookahead () = match !tok_list with
    []       -> None
  | (h::t)   -> Some h

let match_tok a = match !tok_list with
    (h::t) when a = h     -> tok_list := t
  | _                     -> raise (ParseError "bad match")

let rec parse_S () =
  match (lookahead()) with
    Some Tok_a            -> match_tok Tok_a
  | Some Tok_t            -> match_tok Tok_t; parse_S (); parse_E ()
  | _                     -> raise (ParseError "parse error")

and parse_E () =
  match (lookahead()) with
  | Some Tok_e            -> match_tok Tok_e; parse_S (); match_tok Tok_d;
  | Some Tok_d            -> match_tok Tok_d
  | _                     -> raise (ParseError "parse error")
```

7. Security [12 pts]

Recall Project 7, in which you were finding and fixing security issues in Spidey-Web, a cross dimensional forum for the different spider-people to communicate with one another. These questions will relate to the work you did there.

A. The initial version of Spidey-Web failed to perform any authentication at all, which meant anyone could pretend to be any user on the system. One of your friends added the following authentication function. It is called with `user` and `passwd`, both of which are inputs provided by the site's prospective users.

```
def authenticate(user, passwd)
    succ = false
    query = %{
      SELECT Password
      FROM Users
      WHERE User = '#{user}'
    }
    @db.execute(query) do |rec|
        succ = (passwd == rec[0])
    end
    assign_session user if succ
end
```

1. [2 pts] **Circle the part** of this function that is **vulnerable**; *and* **name the vulnerability** category, here:

2. [2 pts] Fix the code by writing in the modifications to the affected lines to the right of the code above. **Strike through the entire problematic line, and write in the replacement just to its right** (but make sure we can still see any parts that you circled, for part 1 above).

B. A new version of Spidey-Web will allow users to have message posting feature. Users are able to post messages addressed to other users. Below is the function that implements this feature, where `Messages` is a new database table that stores message conversations. Code in `main.rb` ensures that any HTTP POST request to `/messages` will trigger the `post_msg` function.

```
def post_msg(user_from, user_to, session, msg)
    query = %{
      UPDATE Messages
      SET Msg = ?
      WHERE User = ?
    }
    @db.execute(query, ["#{user_to}: #{msg}", user_from])
    true
end
```

[6 pts, 3 each] There are (at least) two security problems with this code. **Give their names** here, and briefly **describe where and why they happen, and how to fix them**.

Vulnerability	Why it happens, and how to fix it

C. [2 pts] If Spidey-Web users were accessing the site in an Internet cafe using a standard Wi-Fi network (e.g., in Starbucks), what sort of interactions with the Spidey-Web site are at risk, and how could you address them?

8. Lambda Calculus, Operational Semantics [14 pts]

A. [2 pts] Consider the lambda calculus term (λx.(λy.y) x) ((λw.w) z). **List its free variables**, or write *none* if there aren't any.

B. [4 pts] Recall the grammar of lambda calculus abstract syntax trees:

e ::= *x* | λ*x*.*e* | *e e*

The following is this grammar represented as an OCaml data type, as given in lecture:

```
type expr =
      | Var of string
      | Lam of string * expr
      | App of expr * expr
```

Give the `expr` value for the following lambda expressions (you might want to draw parentheses first, explicitly). For example, for `λx.x` you'd write `Lam ("x", Var "x")`.

1. `λx.x y`

2. `(λx.x) λy.y`

C. [2 pts] The lambda calculus beta reduction rule applies a function to an argument by substituting the latter for the former's parameter. A problem with beta reduction is that it can capture free variables during substitution. **Give an expression for which *alpha conversion* is needed to avoid variable capture** during an imminent beta reduction step.

D. [6 pts] When reducing a lambda calculus expression beta reduction can sometimes be used in more than one place. For example, (λx.(λy.y) x) ((λw.w) z) can reduce three ways:

$(\lambda x.(\lambda \underline{y}.y)\ \underline{x})\ ((\lambda w.w)\ z) \rightarrow (\lambda x.x)\ ((\lambda w.w)\ z)$ since (λy.y) x → x (A)

$(\lambda x.(\lambda y.y)\ x)\ ((\lambda \underline{w}.w)\ \underline{z}) \rightarrow (\lambda x.(\lambda y.y)\ x)\ z$ since ((λw.w) z) → z (B)

$$\begin{tikzcd}(\lambda \underline{x}.(\lambda y.y)\ x) & \underline{((\lambda w.w)\ z)} \arrow[l, bend right] \rightarrow (\lambda y.y)\ ((\lambda w.w)\ z)\end{tikzcd}$$ applying (λx…) to ((λw.w) z) (C)

We can use operational semantics to specify that evaluation can take place within different sub-expressions of an expression. In total, there are four operational rules; here are two:

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1\{e2/x\}}\ \text{(rule 1)} \qquad \frac{e2 \rightarrow e2'}{e1\ e2 \rightarrow e1\ e2'}\ \text{(rule 2)}$$

1. [1 pt] Rule 1 is normal beta reduction (it says to reduce by substituting *e2* for *x* in *e1*). It would apply to example (C), above, where (λx.(λy.y) x) matches the (λ*x*.*e1*) part of the rule and ((λw.w) z) matches the *e2* part. Rule 2 states that if a reduction can happen in the right half of an application, then the whole expression can reduce by rewriting the right half. **Which of the example reductions, (A) or (B), is a use of Rule 2?**

2. [2 pts] **Write the operational semantics rule that covers the example reduction (A or B) you didn't pick for part 1**. (The next part provides a hint.)

3. [3 pts] Recall that operational semantics rules often allow for a direct translation to an interpreter. The function `reduce` below is an interpreter for the lambda calculus, using the data type `expr` given in part B. The code for `reduce` currently has cases that correspond to Rule 1 above, and the rule we asked you to write for part 2. **Add code that corresponds to Rule 2 above**, in the indicated spot (and, *optionally*, any other code you think is important for ensuring applications are handled completely).

```
let rec reduce (e:expr) : expr = match e with
    | App (Lam (x,e1), e2) -> subst e1 x e2 (* Rule 1 shown above *)
    | Lam (x,e1) -> let e1' = reduce e1 in Lam(x,e1')
    | App (e1,e2) -> (* FILL IN for Rule 2 *)
```