CMSC330 Spring 2018 Final Exam

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

____

Instructions

- The exam has 18 pages (front and back); 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 120 points, so allocate 60 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	Lambda Calculus	/8
3	OCaml	/28
4	Ruby	/13
5	Rust	/17
6	Regexps, FAs, CFGs	/20
7	Parsing	/10
8	Operational Semantics	/6
9	Security	/10
	TOTAL	/120

1. PL Concepts [8 pts]

A. [6 pts] Circle T (true) or F (false) for each of the following statements (1 point each)

1. T / F If two OCaml modules implement the same abstract type, values from one module can be used by the functions defined in the other.

2. T / F A type-safe programming language is one in which all well-defined programs (i.e., those which can't "go wrong") are well-typed.

3. T / F In Rust, the borrower's lifetime must not outlast the owner's.

4. T / F OCaml does not require declaring variables with types, which means it is dynamically typed.

5. T / F Reference counting can be used to reclaim cyclic garbage.

6. T / F The OCaml compiler can convert a tail-recursive method into a loop, preventing stack overflow.

B. [2 pts] Multiple Choice (1 point each)

1. Which of these errors is *still* possible in a language with garbage collection?
   a) Memory leak
   b) Dangling pointer
   c) Double free
   d) Use after free
2. Which of the following OCaml features does *not* have an equivalent in Rust?
   a) Records
   b) Variant types (aka datatypes)
   c) Garbage collection
   d) Parametric polymorphism

2. Lambda Calculus [8 pts]

A. [1 pt] Circle all the free variables in the following expression.

λx. (λs. s q) (λq. q x)

B. [3 pts] Reduce the following expression using the call-by-name (CBN) evaluation strategy. Show all alpha conversions and beta reductions for full points.

(λx. x a b) ((λx.y) a)

C. [4 pts] Prove that Y F is *equivalent* to F (Y F) where Y is the fixpoint combinator, given below, and F is an arbitrary lambda term. Two terms are equivalent if they beta-reduce to the same term. For example, (λf.f) x y and (λf.f y) x are equivalent because they both beta-reduce to x y. (*Hint*: beta-reduce Y F and check for a close equivalence to F (Y F) after each step.)

Y = λf.(λx.f (x x)) (λx.f (x x))

3. OCaml [28 pts]

A. Write the types of the following OCaml expressions. If the expression doesn't type check, write "type error" and briefly give a reason why.

1. [2 pts]

```
fun a b c -> c::(a +. 2.0, b ^ "cat")
```

2. [2 pts]

```
fun x y -> y x x
```

B. Provide expressions, without using type annotations, that have the following types. Use of the List module is permitted.

1. [2 pts]

```
(int -> int) -> int
```

2. [2 pts]

```
‘a list -> ‘a list -> (‘a * ‘a)
```

C. What is 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.) We have provided the implementation of `fold` for your convenience.

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

1. [3 pts]

```
let reimann_sum lst_of_lst =
  fold (fun acc lst ->
    acc + fold (fun a h -> a * h) 1 lst) 0 lst_of_lst
in reimann_sum [[2;4]; [1;9]]
```

2. [3 pts]

```
type production = {nonterminal: string; conjunction: string list}
type grammar = production list

let rec help lst = match lst with
 | [] -> ""
 | [x] -> x ^ "\n"
 | h::t -> h ^ " | " ^ (help t)

let rec grammify gram = match gram with
  | [] -> ""
  | {nonterminal = s; conjunction = lst}::t ->
  s ^ " -> " ^ (help lst) ^ (grammify t);;

grammify [{nonterminal = "A"; conjunction = ["B";"ab"]};
          {nonterminal = "B"; conjunction = ["A";"cd"]}];;
```

D. Recall that a multiset is a set that may contain duplicates (the number of duplicates of an element is called its *multiplicity*). One possible implementation is to use regular lists:

```
type 'a bad_mset = 'a list
```

An example in this representation is `['a';'a';'b';'b';'c';'b';'b']`, where the multiplicity of `'a'` is 2, the multiplicity of `'b'` is 4, and the multiplicity of `'c'` is 1. This representation is memory-inefficient if we have a multiset with many duplicates. A multiset in which the multiplicity of `'a'` is 1000 would contain 1000 copies of `'a'`, wasting memory.

1. [3 pts] Write the type for a more efficient representation of a multiset whose storage is proportional to the number of distinct elements, rather than the total count.

```
type 'a mset =
```

2. [5 pts] Implement the function `mult xs e` that returns the multiplicity of element `e` in the multiset `xs` (and zero if it doesn't exist). Use your `mset` type.

```
let rec mult (xs : 'a mset) (e : 'a) : int =
```

3. [6 pts] Implement the function `sum xs ys` that returns the *sum* of two multisets. If `zs = sum xs ys`, then `zs` has its elements drawn from `xs` and `ys` (i.e., `zs` is their union) and the multiplicity of an element in `zs` is the sum of the multiplicities of that element in `xs` and `ys`. Use your `mset` type from problem (D.1). You may also use your `mult` from problem (D.2).

```
let rec sum (xs : 'a mset) (ys : 'a mset) : ('a mset) =
```

4. Ruby [13 pts]

Thanos is taking 330 this semester and doesn't have a passing grade going into the final. He didn't do this year's Rust project because the TAs weren't very helpful. He has devised a plan to discover all the TAs who don't know Rust and capture them all as punishment! As his assistant, you will help him write a Ruby class `RustIdentifer` used to identify TAs and assemble this list. Here are the methods of this class:

`add(filename):` [4 pts] opens a file and stores the information within a data structure of your choosing. Each line of the **`filename`** consists of a TA's name (all alphabetic letters, with the first being uppercase), followed by a comma, followed by either "Y" or "N" depending on whether the TA knows Rust or not. If there are duplicate names, occurrences past the first one should be skipped. Illegal TA names should be skipped too. An example input file is as follows:

```
Omar,Y
Stephen,N
Timothy,N
Omar,N
Not-a-TA,Y
```

In the above example, Omar knows Rust (from the first occurrence, as the second is skipped), and you should skip Not-a-TA (since it has non-alphabetic characters).

`knowsNotRust():` [4 pts] returns an array of TAs who do not know Rust.

`captureTAs(names):` [5 pts] will traverse the structure and delete any TAs in the structure whose name is in the input list **`names`**.

Note that you may or may not need to fill out the `initialize()` function, depending on your approach.

```
class RustIdentifer
      def initialize()


      end

      def add(filename)
            File.foreach(filename) do |line|










            end
      end

      def knowsNotRust()











      end

      def captureTAs(names)









      end
end
```

5. Rust [17 pts]

The code snippet below is for part (a) and (b).

```
1  fn main() {
2       let nums = vec![1,2,3,4];
3       let sum = sum(&nums);
4       let prod = prod(nums);
5       println!("sum: {}, prod: {}", sum, prod);
6  }
7
8  fn sum(lst: &Vec<i32>) -> i32 {
9       lst.iter().fold(0,|a,h| a + h)
10 }
11
12 fn prod(lst: Vec<i32>) -> i32 {
13      lst.iter().fold(1,|a,h| a * h)
14 }
```

A. [2 pts] Which of the above functions has ownership of **nums** at any point during execution?

B.[2 pts] At which line will **nums** be freed? _____________

The code snippet below is for part (c)

```
1  fn main() {
        // Mistake 1 is here
2       let x = 1;
3       while x < 10 {
4             println!("x is {}",x);
5             x += 1;
6       };
        // Mistake 2 is here
7       let y = &x;
8       if y == 10 {
9             println!("y is 10")
10      }
```

```
        // Mistake 3 is here
11      let mut v = vec![1,2,3,4];
12      let g = v.get(1).unwrap();
13      *g += 1;
14      println!("g is {}",g);
        // Mistake 4 is here
15      let a = Some(String::from("330"));
16      let b = &a;
17      match b {
18            &Some(s) => println!("b is a reference to Some({})", s),
19            &None => println!("b is a reference to None")
20      };
        // Mistake 5 is here
21      let mut f = String::from("a string");
22      let mut mrf = &f;
23      mrf.push_str("modify");
24      println!("mrf is a mutable reference to {}", mrf);
35  }
```

C. [6 pts] The above code has 5 mistakes in it. There are no syntax errors; all errors are type errors or borrow errors. Moreover, you can be sure that the compiler never employs the *deref coercion* in this code. Each mistake is independent from the others, and is on a single line.

Find any 3 of the 5 errors. **Write the line number of the error and rewrite the line to be fixed**. Your change should not change the intended functionality of the line (e.g., you can't just delete it). You will not receive credit for finding any mistakes beyond the first 3.

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<T>: fn unwrap(self) -> T`
Moves the value v out of the `Option<T>` if it is `Some(v)`.

`String: fn push_str(&mut self, string: &str)`
Appends the string slice `str` onto the end of this `String`.

Line Number	Rewritten Line, with Fix

D.[7 pts] Write a function **`max`** that takes an immutable reference to a generic slice and returns a tuple of the max value in the slice, and the index of that value. For reference, you can get an `Iterator` from a slice, via a call to its `iter()` method, and the `enumerate()` method of such an iterator returns an `Iterator` of (index, value) pairs.

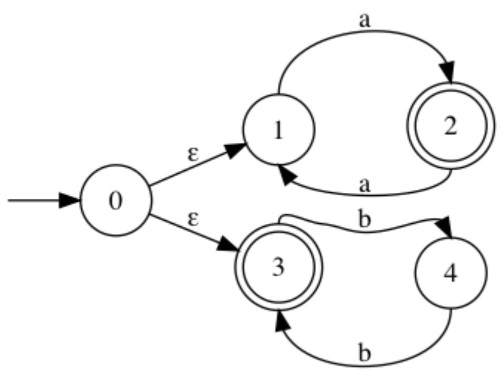
```
fn max<T: PartialOrd + Eq> (lst: &[T]) -> (usize, &T) {
   assert!(!lst.is_empty());















}
```

6. Regexps, FAs, CFGs [20 pts]

A. [4 pts] Draw an NFA for the following regular expression: `((ab)*b*) | b`

B. [4 pts] Give a regular expression for all strings over the alphabet **`{a,b,c}`** that contain exactly two **a**'s (and any number of **b**'s or **c**'s).

C. [3 pts] Describe (as either a regex or in plain English) the strings that the following NFA accepts:



D. [5 pts] Convert the above NFA to a DFA.

E. [4 pts] Write a context free grammar that generates the language:

$a^x b^y c^z$, where $y = x + z$ and $x > 0$ and $z > 0$

7. Parsing [10 pts]

A. [4 pts] **Fix the following context free grammar** so that it can be parsed by a recursive descent parser (which, we remind you, is a kind of predictive parser).

S → bcA | bdA
A → Aa | a

B. [6 pts] Given the following code for a parser written in OCaml, **write the grammar that is being parsed**. This code assumes tokens are `strings`, the `lookahead()` function returns the topmost token x as `Some x`, and `None` if there are no tokens left; the `match_tok(x)` function consumes the topmost token if it matches x and raises an error otherwise; and `error()` raises an error unconditionally.

```
let rec parse_A () =
  if (lookahead () = Some "a") then
      (match_tok("a");  parse_S ();  parse_A ())
  else if (lookahead () = Some "f") then
      match_tok("f")
  else
      error()

and parse_S () =
  if (lookahead () = Some "a" || lookahead () = Some "f") then
      (parse_A(); match_tok("b"); match_tok("c"))
  else if (lookahead () = Some "d") then
      (match_tok("d"); parse_S())
  else ()
```

8. Operational Semantics [6 pts]

A. [3 points] Given the following SmallC operational semantics rules from project 3, **complete the derivation** of the statement below:

Int $\dfrac{}{A;\, n \Rightarrow n}$

Assign-Int $\dfrac{A(x) = n \qquad A;\, e \Rightarrow n_1}{A;\, x = e \Rightarrow A[x \mapsto n_1]}$

Bool-True $\dfrac{}{A;\, true \Rightarrow true}$

Bool-False $\dfrac{}{A;\, false \Rightarrow false}$

If-True $\dfrac{A;\, e \Rightarrow true \quad A;\, s_1 \Rightarrow A_1}{A;\, if\ (e)\ \{s_1\}\ \{s_2\} \Rightarrow A_1}$

If-False $\dfrac{A;\, e \Rightarrow false \quad A;\, s_2 \Rightarrow A_2}{A;\, if\ (e)\ \{s_1\}\ \{s_2\} \Rightarrow A_2}$

$$\dfrac{}{\bullet[x \mapsto 0];\ \text{if (false) \{x = 1\} \{x = 2\}} \Rightarrow \bullet[x \mapsto 2]}$$

B. [3 points] The following rule is part of the operational semantics for SmallC:

$$\dfrac{\begin{array}{l} A\ ;\ e \Rightarrow true \\ A\ ;\ s \Rightarrow A_1 \\ A_1\ ;\ while\ (e)\ \{\, s \,\} \Rightarrow A_2 \end{array}}{A\ ;\ while\ (e)\ \{\, s \,\} \Rightarrow A_2}$$

Explain this rule, in words. Your explanation should be something of the variety *if under environment A expression e evaluates to … then …* etc.

9. Security [10 pts]

A**.** [2 pts] Why do secure websites (e.g., Google, Stark Industries) store the hash of your password instead of the password itself (i.e. what scenario(s) does this protect against)? Limit your answer to 2-3 sentences.

B. [2 pts] **<u>Circle</u>** TRUE or FALSE for each statement:

1) Stored XSS attacks are usually carried out with specially crafted URLs.

TRUE FALSE

2) Cookies are pieces of data stored in the browser sometimes used to identify an authenticated client to a server.

TRUE FALSE

C. StarkServe is a new service at Stark Industries. The service is a simple Ruby server that acts as a remote Ruby shell. It allows you to send it a Ruby command and it sends back the output from executing it, similar to a what a stateless top-level Ruby interpreter (irb) would do. The server works by passing your command its method `interpret` (shown below). This method runs the command and returns its output. StarkServe is running on a server with sensitive corporate secrets that it doesn't want users to be able to read, modify or destroy.

```
def interpret(data)
      #returns array of size 3 containing the standard output,
      #standard error, and exit status
      stdout, stderr, status = Open3.capture3("ruby -e \"#{data}\"")

      if status == 0
            stdout
      else
            stderr
      end
end
```

(go to next page)

Example usage (`input` in normal case, *`output` in italics*)**:**

```
$ nc 192.168.239.245 1324
Welcome to StarkServe! You may send any input and StarkServe will interpret
it for you!
puts \“hello world\”
Output: hello world
if false then puts \"true\" else puts \"false\" end
Output: false
```

[2 pts] Which class of vulnerability (or vulnerabilities) is StarkServe vulnerable to and why?

[2 pts] Give or describe input that would exploit a StarkServe vulnerability.

[2 pts] State two techniques that could be used to fix the vulnerability. Be specific to this case, pointing to the code if you think it will help.

END OF EXAM