# CMSC330 - Organization of Programming Languages Summer 2023 - Final

CMSC330 Course Staff University of Maryland Department of Computer Science

Name: \_\_\_\_\_

UID: \_\_\_\_\_

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination

Signature: \_\_\_\_\_

#### **Ground Rules**

- · You may use anything on the accompanying reference sheet anywhere on this exam
- Please write legibly. If we cannot read your answer you will not receive credit
- You may not leave the room or hand in your exam within the last 10 minutes of the exam
- If anything is unclear, ask a proctor. If you are still confused, write down your assumptions in the margin

Question	Points
Q1	10
Q2	10
Q3	15
Q4	12
Q5	6
Q6	4
Q7	10
Q8	13
Total	80

#### **Problem 1: Language Concepts**

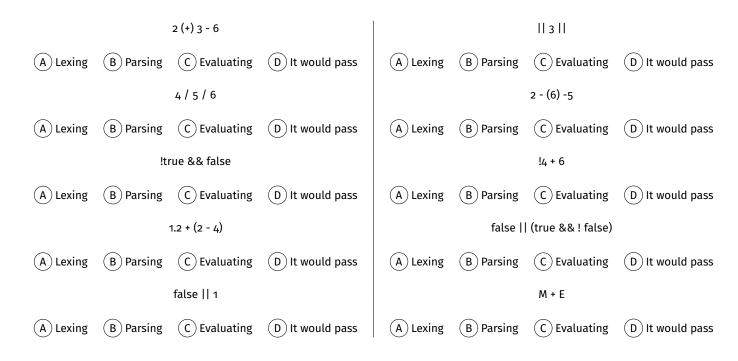
x:'a &i32, y:'b &i32 have the same type	True T	False F
Operational Semantics is to evaluator as CFG is to parser	T	F
The reference counting garbage collection strategy uses less space than the stop and copy one (on average)	T	F
If you cannot eagerly evaluate, then you also cannot lazily evaluate a $\lambda$ -calculus expression	T	F
Expressions and Statements can be used interchangeably	(T)	<b>(F</b> )

#### **Problem 2: Interpreters**

Consider the following Grammar and assume semantics follows Python's behavior

 $\begin{array}{rcl} E \Rightarrow & M + E \mid M \mid \mid E \mid M - E \mid M \\ M \Rightarrow & N * M \mid N \& \& M \mid N / M \mid N \\ N \Rightarrow & !P \mid P \\ P \Rightarrow & n \in \mathbb{N} \mid true \mid false \mid (E) \end{array}$ 

Which step of the interpreter (if any) would the following fail at?



[Total 10 pts]

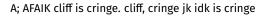
[Total 10 pts]

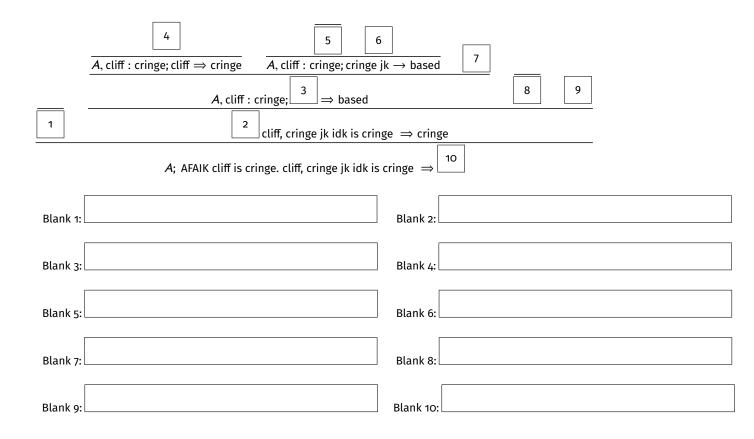
### **Problem 3: Operational Semantics**

Kids and their weird slang! How is an old man like Cliff supposed to keep up? Consider the following rules for CringeCode, which uses "based" for true and "cringe" for false with Python as the Metalanguage:

Rule 1: 
$$fightarrow basedRule 2:  $fittarrow cringeRule 1:  $fitarrow basedRule 2:  $fitarrow cringeRule 3:  $fitarrow cringe in a particular structureRule 2:  $fitarrow cringe in a particular structureRule 3:  $fitarrow cringe in a particular structureRule 2:  $fitarrow cringe in a particular structureRule 3:  $fitarrow cringe in a particular structureRule 2:  $fitarrow cringe in a particular structureRule 3:  $fitarrow cringe in a particular structureRule 2:  $fitarrow cringe in a particular structureRule 3:  $fitarrow cringe in a particular structureRule 2:  $fitarrow cringe in a particular structureRule 3:  $fitarrow cringe in a particular structureRule 4:  $fitarrow cringe in a particular structureRule 5:  $fitarrow cringe in a particular structureRule 6:  $fitarrow cringe in a particular structureRule 5:  $fitarrow cringe in a particular structureRule 6:  $fitarrow cringe in a particular structureRule 7:  $fitarrow cringe in a particular structureRule 8:  $fitarrow cringe in a particular structure$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

Using the above rules, prove the following sentence evaluates to cringe:





#### **Problem 4: Rust Features**

[Total 12 pts]

1 2	fn main(){ <b>let</b> m = String::from("Hello");	Ownership If there is no owner, write "NONE".
3 4 5	<pre>let t = String::from("World"); let mut z = String::from("CMSC330"); { let w = m;</pre>	Who is the owner of "Hello" immediately after line 6 is run?
6 7	{ <b>let</b> c = foo(w,t); <b>let</b> d = bar(&z,&z,&c);	Who is the owner of "World" immediately after line 14 is run?
8 9 10	z = String::from(d); } };	Lifetimes
11 12	println!("{z}") }	What is the last line executed before "Hello" dropped?
13 14 15 16	fn foo(a:String, b: String) -> String{ <b>if</b> a.len() > b.len() {a} <b>else</b> {b} }	What is the last line executed before "World" dropped?
17 18 19	fn bar<'a,'b>(x:&'a str, y:&'b str,	At what line does z's lifetime end?
20 21 22	p:&'a str) -> &'a str{ if x == y {x} else {p} }	At what line does c's lifetime end?

### Problem 5: OCaml Typing

Given the following type, write an expression that matches that type. You may not use type annotations, and all pattern matching must be exhaustive.

(a)'a list -> ('b list -> 'a -> 'b list) -> 'b list -> int

Given the expression, write down its type.

(b) fun a b c -> (map c a)::[[1]]

## **Problem 6: Lambda Calculus**

Perform a single  $\beta$ -reduction using the eager (call by value) evaluation strategy on the outermost expression. If you cannot reduce it, write **Beta Normal Form**. Do **not**  $\alpha$ -convert your final answer.

(a)  $(x \lambda x. x x)(\lambda x. x x)$ 

[Total 6 pts]

[3 pts]

[3 pts]

[Total 4 pts]

[2 pts]

Perform a single  $\beta$ -reduction using the lazy (call by name) evaluation strategy on the outermost expression. If you cannot reduce it, write **Beta Normal Form**. Do **not**  $\alpha$ -convert your final answer.

(b)  $(\lambda x. x y x)((\lambda x. (x x)) x)$ 

[2 pts]

## **Problem 7: Ocaml Programming**

[Total 10 pts]

Recall the move function for a FSM. It takes in a character, a state, and a FSM, and it returns a list of states. Let's modify this a little bit. Given a partial FSM, you will move on all states with the symbol provided. Your return type will be (int \* int list) list, where the int is the state you moved on, and the int list is the states you can move to. You **may not** use the rec keyword but you can make non-recursive helper functions.

```
type partial_fsm = (int list * (int * string * int) list);
(* int list is state list.
(int * string * int) list is transition list.
let states = [1;2;3;4] in
let trans = [(1,"a",2);(1,"a",3);(2,"a",4)] in
let pfsm = (states,trans) in
move_all pfsm "a" => [(1,[2;3]);(2,[4]);(3,[]);(4,[])]
Order does not matter *)
let move_all pfsm symbol =
```

### **Problem 8: Rust Programming**

Write a lexer in Rust for the grammar: (E -> E + E | E - E | n) where n is any integer. Your tokens are "Number", "Add", and "Sub". For example lexer("3 + 2 - 1") returns a vector that looks like ["Number", "Add", "Number", "Sub", "Number"]. **Note**: To separate negative integers and subtraction, there will be a space between numbers and the subtraction symbol. For example:

```
lex("3 - 4") == ["Number", "Sub", "Number"]
lex("3 -4") == ["Number", "Number"]
```

```
fn lex(sentence:&str) -> Vec<&str>
```

## **Cheat Sheet**

## Rust

```
// Vectors
let vec = Vec::new();
                                                       // looping
let mut vec1 = Vec!({1,2,3,4]);
                                                       while guard {...}
vec1[2] // returns 3
                                                       while true{ ... }
vec1.push(5); // vec1 becomes [1,2,3,4,5]
                                                       // will loop until the guard is false or until
                                                       // a break statement
let x = vec1.pop(); //x = 5, vec1 = [1,2,3,4]
vec1[0] = vec1[0] + 1; // vec1 = [2,2,3,4]
                                                       for x in iterator {...}
                                                       for i in 0..5 {...}
let vec_slice = &vec1[1..3];
                                                       for &x in vec![1,2,3].iter() { ... }
                                                       // will iterate through an iterator
enum Name{
                                                       // Many types like Vectors have an iterator
    Type1,
                                                       // method or similar
    Type2: String
}
                                                       // Strings and &str
                                                       let s = String::from("string");
struct User {
                                                       let s1 = "String";
    active: bool,
                                                       // s is stored on heap
    username: String,
                                                       // s1 is stored on stack
}
                                                       let mut s2 = String::from("Hello");
// regex in rust
                                                       s2.push_str(", World!");
Regex::new(&str)
                                                       // s2 is now "Hello, World!"
let re = Regex::new(r"I am (\d+) years old");
// Compiles a regular expression. Once compiled,
                                                       //slices and substrings
// it can be used repeatedly to search, split or
                                                       let a = s2[1..3]; // a = "el";
// replace text in a string. Returns a Result Object
                                                       // string methods
re.is_match(&str)
                                                       s.len()
assert!(re.is_match("I am 19 years old"));
                                                       //gets length of string.
// returns true iff there is a match anywhere
// in the string. Returns false otherwise
                                                       s.insert(i32,char);
                                                       s2.insert(0,'A');
re.find(&str)
                                                       //s2 is now "AHello, World!"
let mat = re.find("I am 19 years old);
assert_eq!(mat.start(), 5);
                                                       s.insert_str(i32,&str);
assert_eq!(mat.end(), 7);
                                                       s2.insert(1," new ");
// Returns the start and end byte range of the
                                                       //s2 is now "A new Hello, World!"
// leftmost-first match in text. If no match exists,
// then None is returned.
                                                       s.chars()
                                                       // returns an iterator over the string going
re.captures(&str)
                                                       // character by character
let cap = re.captures("I am 19 years old");
let age = cap.get(1);
                                                       //to throw an error
assert_eq!(age, "19");
                                                       panic!("error msg");
// returns the capture groups of a regex. If no
// match is found, returns None
```

## Regex

*	zero or more repetitions of the preceding character or group
+	one or more repetitions of the preceding character or group
?	zero or one repetitions of the preceding character or group
•	any character
$r_1   r_2$	$r_1$ or $r_2$ (eg. a b means 'a' or 'b')
$[r_1r_2r_3]$	<i>r</i> <sub>1</sub> or <i>r</i> <sub>2</sub> or <i>r</i> <sub>3</sub> (eg. [abc] is 'a' or 'b' or 'c')
[^ <i>r</i> <sub>1</sub> ]	anything except $r_1$ (eg. [^abc] is anything but an 'a', 'b', or 'c')
$[r_1 - r_2]$	range specification (eg. [a-z] means any letter in the ASCII range of a-z)
{n}	exactly n repetitions of the preceding character or group
{n,}	at least n repetitions of the preceding character or group
{m,n}	at least m and at most n repetitions of the preceding character or group
^	start of string
\$	end of string
( <i>r</i> <sub>1</sub> )	capture the pattern $r_1$ and store it somewhere (match group in Python)
\d	any digit, same as [0-9]
\s	any space character like \n, \t, \r, \f, or space

# **Ocaml Map and Fold**

## Grammars

<pre>let rec map f l = match l with [] -&gt; []</pre>						
h::t -> (f h)::(map f t)	Regex			$\lambda$ -calc		
	R	$\rightarrow$	Ø	е	$\rightarrow$	x
<pre>let rec fold_l f a l = match l with</pre>			$\sigma$		I	λx.e
[] -> a			$\epsilon$			e e
h::t->fold_lf(fah)t			RR			
			R R			
<pre>let rec_fold_r f l a = match l with</pre>		I	R *			
[] -> a						
h::t->fh (fold_rfta)						

# Lambda Calc and Opsem Encodings

We will give you the encodings that you will need. They may or may not look like/include the following:

> $\lambda x.\lambda y.x = \text{true}$   $\lambda x.\lambda y.y = \text{false}$  $e_1 \ e_2 \ e_3 = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$

We will give you the opsem rules that you will need. They may or may not look like/include the following:

$$\overline{n \to n}$$

$$\frac{A; e_1 \Rightarrow v_1 \quad v_2 \text{ is not } v_1}{A; !e_1 \Rightarrow v_2}$$

$$\frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 \text{ is } v_1 + v_2}{A; e_1 + e_2 \Rightarrow v_3}$$