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

[Total 10 pts]

	True	False
`x:'a &i32, y:'b &i32` have the same type	T	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 λ-calculus expression	T	F
Expressions and Statements can be used interchangeably	T	F

Problem 2: Interpreters

[Total 10 pts]

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

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

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

2 (+) 3 - 6

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

4 / 5 / 6

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

!true && false

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

1.2 + (2 - 4)

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

false || 1

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

|| 3 ||

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

2 - (6) -5

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

!4 + 6

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

false || (true && ! false)

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

M + E

(A) Lexing (B) Parsing (C) Evaluating (D) It would pass

Problem 3: Operational Semantics

[Total 15 pts]

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: $\dfrac{}{\text{based} \Rightarrow \text{based}}$ Rule 2: $\dfrac{}{\text{cringe} \Rightarrow \text{cringe}}$

Rule 3: $\dfrac{A;\ e_1 \Rightarrow v_1 \qquad v_2\ ==\ not\ v_1}{A;\ e_1\ \text{jk} \Rightarrow v_2}$ Rule 4: $\dfrac{A;\ e_1 \Rightarrow v_1 \qquad A;\ e_2 \Rightarrow v_2 \qquad v_3\ ==\ v_1\ \text{or}\ v_2}{A;\ e_1,\ e_2\ \text{idk} \Rightarrow v_3}$

Rule 5: $\dfrac{A, x : v\ (\text{x})\ =\ v}{A, x : v;\ \text{x} \Rightarrow v}$ Rule 6: $\dfrac{A;\ e_1 \Rightarrow v_1 \qquad A, \text{x} : v_1;\ e_2 \Rightarrow v_2}{A;\ \text{AFAIK}\ x\text{'s}\ e_1 .\ e_2 \Rightarrow v_2}$

Rule 7: $\dfrac{A;\ e_1 \Rightarrow v_1 \qquad A;\ e_2 \Rightarrow v_2 \qquad v_1\ ==\ v_2}{A;\ e_1 \text{is}\ e_2 \Rightarrow \text{based}}$ Rule 8: $\dfrac{A;\ e_1 \Rightarrow v_1 \qquad A;\ e_2 \Rightarrow v_2 \qquad v_1 != v_2}{A;\ e_1 \text{is}\ e_2 \Rightarrow \text{cringe}}$

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

A; AFAIK cliff is cringe. cliff, cringe jk idk is cringe

$$\dfrac{\dfrac{}{\boxed{1}} \qquad \dfrac{\dfrac{\dfrac{\boxed{4}}{A, \text{cliff} : \text{cringe}; \text{cliff} \Rightarrow \text{cringe}} \qquad \dfrac{\dfrac{}{\boxed{5}} \quad \boxed{6}}{A, \text{cliff} : \text{cringe}; \text{cringe jk} \longrightarrow \text{based}} \qquad \boxed{7}}{A, \text{cliff} : \text{cringe}; \boxed{3} \Rightarrow \text{based}} \qquad \dfrac{}{\boxed{8}} \qquad \boxed{9}}{\boxed{2}\ \text{cliff, cringe jk idk is cringe} \Rightarrow \text{cringe}}}{A;\ \text{AFAIK cliff is cringe. cliff, cringe jk idk is cringe} \Rightarrow \boxed{10}}$$

Blank 1: ______ Blank 2: ______

Blank 3: ______ Blank 4: ______

Blank 5: ______ Blank 6: ______

Blank 7: ______ Blank 8: ______

Blank 9: ______ Blank 10: ______

Problem 4: Rust Features

[Total 12 pts]

```
1  fn main(){
2      let m = String::from("Hello");
3      let t = String::from("World");
4      let mut z = String::from("CMSC330");
5      { let w = m;
6        { let c = foo(w,t);
7          let d = bar(&z,&z,&c);
8          z = String::from(d);
9        }
10     };
11     println!("{z}")
12 }
13
14 fn foo(a:String, b: String) -> String{
15     if a.len() > b.len() {a} else {b}
16 }
17
18 fn bar<'a,'b>(x:&'a str,
19               y:&'b str,
20               p:&'a str) -> &'a str{
21     if x == y {x} else {p}
22 }
```

Ownership

If there is no owner, write "NONE".

Who is the owner of `"Hello"` immediately after line 6 is run?

Who is the owner of `"World"` immediately after line 14 is run?

Lifetimes

What is the last line executed before `"Hello"` dropped?

What is the last line executed before `"World"` dropped?

At what line does z's lifetime end?

At what line does c's lifetime end?

Problem 5: OCaml Typing

[Total 6 pts]

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` [3 pts]

Given the expression, write down its type.

(b) `fun a b c -> (map c a)::[[1]]` [3 pts]

Problem 6: Lambda Calculus

[Total 4 pts]

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

(a) $(x\ \lambda x.\ x\ x)(\lambda x.\ x\ x)$ [2 pts]

Perform a single β-reduction using the lazy (call by name) evaluation strategy on the outermost expression. If you cannot reduce it, write **Beta Normal Form**. Do **not** α-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

[Total 13 pts]

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 seperate 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();
let mut vec1 = Vec!({1,2,3,4]);
vec1[2] // returns 3
vec1.push(5); // vec1 becomes [1,2,3,4,5]

let x = vec1.pop(); //x = 5, vec1 = [1,2,3,4]
vec1[0] = vec1[0] + 1; // vec1 = [2,2,3,4]

let vec_slice = &vec1[1..3];

enum Name{
    Type1,
    Type2: String
}

struct User {
    active: bool,
    username: String,
}

// regex in rust
Regex::new(&str)
let re = Regex::new(r"I am (\d+) years old");
// Compiles a regular expression. Once compiled,
// it can be used repeatedly to search, split or
// replace text in a string. Returns a Result Object

re.is_match(&str)
assert!(re.is_match("I am 19 years old"));
// returns true iff there is a match anywhere
// in the string. Returns false otherwise

re.find(&str)
let mat = re.find("I am 19 years old);
assert_eq!(mat.start(), 5);
assert_eq!(mat.end(), 7);
// Returns the start and end byte range of the
// leftmost-first match in text. If no match exists,
// then None is returned.

re.captures(&str)
let cap = re.captures("I am 19 years old");
let age = cap.get(1);
assert_eq!(age, "19");
// returns the capture groups of a regex. If no
// match is found, returns None
```

```
// looping
while guard {...}
while true{ ... }
// will loop until the guard is false or until
// a break statement

for x in iterator{...}
for i in 0..5 {...}
for &x in vec![1,2,3].iter() { ... }
// will iterate through an iterator
// Many types like Vectors have an iterator
// method or similar

// Strings and &str
let s = String::from("string");
let s1 = "String";
// s is stored on heap
// s1 is stored on stack

let mut s2 = String::from("Hello");
s2.push_str(", World!");
// s2 is now "Hello, World!"

//slices and substrings
let a = s2[1..3]; // a = "el";

// string methods
s.len()
//gets length of string.

s.insert(i32,char);
s2.insert(0,'A');
//s2 is now "AHello, World!"

s.insert_str(i32,&str);
s2.insert(1," new ");
//s2 is now "A new Hello, World!"

s.chars()
// returns an iterator over the string going
// character by character

//to throw an error
panic!("error msg");
```

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
$r_1 \mid r_2$	r_1 or r_2 (eg. a|b means 'a' or 'b')
$[r_1 r_2 r_3]$	r_1 or r_2 or r_3 (eg. [abc] is 'a' or 'b' or 'c')
$[\hat{} r_1]$	anything except r_1 (eg. [ˆabc] is anything but an 'a', 'b', or 'c')
$[r_1\text{-}r_2]$	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
ˆ	start of string
$	end of string
(r_1)	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

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

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

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

Grammars

Regex

$$
\begin{array}{rcl}
R & \rightarrow & \emptyset \\
 & \mid & \sigma \\
 & \mid & \epsilon \\
 & \mid & RR \\
 & \mid & R|R \\
 & \mid & R^*
\end{array}
$$

λ-calc

$$
\begin{array}{rcl}
e & \rightarrow & x \\
 & \mid & \lambda x.e \\
 & \mid & e\ e
\end{array}
$$

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:

$$
\begin{array}{rcl}
\lambda x.\lambda y.x & = & \text{true} \\
\lambda x.\lambda y.y & = & \text{false} \\
e_1\ e_2\ e_3 & = & \text{if } e_1 \text{ then } e_2 \text{ else } e_3
\end{array}
$$

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

$$\frac{}{\mathrm{n} \rightarrow \mathrm{n}}$$

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

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