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

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>10</td>
</tr>
<tr>
<td>Q2</td>
<td>10</td>
</tr>
<tr>
<td>Q3</td>
<td>15</td>
</tr>
<tr>
<td>Q4</td>
<td>12</td>
</tr>
<tr>
<td>Q5</td>
<td>6</td>
</tr>
<tr>
<td>Q6</td>
<td>4</td>
</tr>
<tr>
<td>Q7</td>
<td>10</td>
</tr>
<tr>
<td>Q8</td>
<td>13</td>
</tr>
<tr>
<td>Total</td>
<td>80</td>
</tr>
</tbody>
</table>
**Problem 1: Language Concepts**

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

**Problem 2: Interpreters**

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

\[
E \Rightarrow M + E \mid M \| E \mid M - E \mid M \\
M \Rightarrow N * M \mid N \&\& M \mid N / M \mid N \\
N \Rightarrow !_P \mid P \\
P \Rightarrow n \in \mathbb{N} \mid true \mid false \mid (E)
\]

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
- \(1/4 + 6\)
  - 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

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: based ⇒ based

Rule 2: cringe ⇒ cringe

Rule 3: A; e₁ ⇒ v₁ v₂ == not v₁

Rule 4: A; e₁ ⇒ v₁ A; e₂ ⇒ v₂ v₃ == v₁ or v₂

Rule 5: A, x : v (x) = v

Rule 6: A, x : v₁; e₂ ⇒ v₂

Rule 7: A; e₁ ⇒ v₁ A; e₂ ⇒ v₂ v₁ == v₂

Rule 8: A; e₁ ⇒ v₁ A; e₂ ⇒ v₂ v₁ != v₂

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

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

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

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

A, cliff : cringe; cliff ⇒ cringe

A, cliff : cringe; cringe jk → based

A, cliff : cringe; cliff ⇒ based

A; AFAIK cliff is cringe. cliff, cringe jk idk is cringe ⇒ 10
Problem 4: Rust Features

```rust
fn main() {
    let m = String::from("Hello");
    let t = String::from("World");
    let mut z = String::from("CMSC330");
    {
        let w = m;
        {
            let c = foo(w, t);
            let d = bar(&z, &z, &c);
            z = String::from(d);
        }
    }
    println!("{z}");
}
```

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?
At what line does z's lifetime end?
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 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 λx.x)x)(λx.x x)
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) \((\lambda x. x y x)((\lambda x. (x x)) x)\)

**Problem 7: Ocaml Programming**

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 \((\text{int} \times \text{int list})\text{ list}\), 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.

```ocaml
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 *

```ocaml
let move_all pfsm symbol =
```
Problem 8: Rust Programming

Write a lexer in Rust for the grammar: \( E \rightarrow E + E \mid E - E \mid n \) where \( n \) is any integer. Your tokens are "Number", "Add", and "Sub". For example \( \text{lexer("3 + 2 - 1") \rightarrow \text{["Number","Add","Number","Sub","Number"]}} \).

**Note:** To separate negative integers and subtraction, there will be a space between numbers and the subtraction symbol.

For example:

\[
\text{lex("3 - 4") == ["Number", "Sub", "Number"]}
\]
\[
\text{lex("3 - 4") == ["Number", "Number"]}
\]

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

// Vectors
let vec = Vec::new();
let mut vec1 = Vec![1, 2, 3, 4];
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

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

for &x in vec![[1, 2, 3]].iter() {...}
// 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
r₁/ r₂ r₁ or r₂ (eg. a|b means 'a' or 'b')
[r₁ r₂ r₃] r₁ or r₂ or r₃ (eg. [abc] is 'a' or 'b' or 'c')
[r₁] anything except r₁ (eg. [^abc] is anything but an 'a', 'b', or 'c')
[r₁-r₂] 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
(r₁) capture the pattern r₁ 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)

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:
λx.λy.x = true
λx.λy.y = false
e₁ e₂ e₃ = if e₁ then e₂ else e₃

Grammars

<table>
<thead>
<tr>
<th>Regex</th>
<th>-calc</th>
</tr>
</thead>
<tbody>
<tr>
<td>R → ∅</td>
<td>e → x</td>
</tr>
<tr>
<td>σ</td>
<td>λx.e</td>
</tr>
<tr>
<td>ε</td>
<td>e e</td>
</tr>
<tr>
<td>RR</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>R*</td>
<td></td>
</tr>
</tbody>
</table>

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

n → n

A; e₁ ⇒ v₁  v₂ is not v₁
A; le₁ ⇒ v₂

A; e₁ ⇒ v₁  A; e₂ ⇒ v₂  v₃ is v₁ + v₂
A; e₁ + e₂ ⇒ v₃