CMSC330 - Organization of Programming Languages
Fall 2025- 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

- Please write legibly. **If we cannot read your answer you will not receive credit.**
- You may use anything on the accompanying reference sheet anywhere on this exam
- You can remove the reference sheet
- If you need extra space, the last page is for scratch work. Make a note for the grader to check the scratch page if you want it graded.
- 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
- Do no take photos of this exam or share this exam in anyway shape or form

Question	Points
P1	10
P2.	12
P3.	10
P4.	10
P5.	8
P6.	6
P7.	8
P8.	8
P9.	8
P10.	20
EC	2
Total	100 + 2

Problem 1: Programming Concepts [Total 10 pts]

(a) True XOR False **True** **False** [6 pts]

Secure programs are programs that have no logic errors	(T)	(F)
Safe Rust mitigates buffer overflow vulnerabilities	(T)	(F)
A Doubly linked list will always be reclaimed using the reference counting garbage collection technique	(T)	(F)
If a lambda calculus expression has a beta normal form when evaluated eagerly, then it also has a BNF when evaluated lazily.	(T)	(F)
The class of languages recognized by finite state machines (FSMs)—the regular languages—is a proper subset of the class of languages generated by context-free grammars (CFGs).	(T)	(F)
`fun x -> fun y -> x (x y)` is the same as `fun x -> fun y -> x x y`	(T)	(F)

(b) Immutability [2 pts]

During OCaml, we stressed the idea of immutability and how this makes things easy to debug and reason about. In fact, many functional languages enforce immutability. That being said, give an example of when mutability is useful.

(c) Laziness [2 pts]

You can use the `lazy` keyword on an expression in OCaml to delay its evaluation until you force it. By default, OCaml evaluates things eagerly. (Hint: `lazy x` is similar to `fun _-> x`, which delays the evaluation of `x`). What is printed out when the following expressions are evaluated?

```
(fun x -> print_string "world") (print_string "hello")
```

```
(fun x -> print_string "world") (lazy (print_string "hello"))
```

Problem 2: Debugging [Total 12 pts]

(a) OCaml Code Tracing [4 pts]

Consider the following incorrect implementation of `Fold`:

```
let rec fold_wrong f a lst = match lst with
[] -> a
|x::xs -> f x (fold_wrong f (f x a) xs)
```

The process of performing a code trace helps debugging. If we start with `fold_wrong (fun a x -> x + a) 2 [1;2;3;4]`, then the **first** recursive call of `fold_wrong` would be `fold_wrong f 3 [2;3;4]`. What are the values of `a,lst,x,xs` **during** the **third** recursive call?

a: _____ *lst*: _____

x: _____ *xs*: _____

(b) Evaluation [2 pts]

What is the result of calling `fold_wrong (fun a x -> x + a) 2 [1;2;3;4]`?

`fold_wrong (fun a x -> x + a) 2 [1;2;3;4]`: _____

(c) Project 3 [6 pts]

Consider the following `Nfa` and `transition` type from project 3:

```
type ('q, 's) nfa_t = {
  sigma: 's list;
  qs: 'q list;
  q0: 'q;
  fs: 'q list;
  delta: ('q, 's)  list;
}
type ('q, 's) transition = {
  input: 's option;
  states: 'q * 'q;
}
```

Consider the following incorrect implementation of `move`:

```
('q, 's) nfa_t -> 'q -> 's option -> 'q list
let rec move nfa state symbol =
  fold_left
    (fun a trans ->
1     let (src,dest) = trans.states in
2     if src = state then
3         dest
4     else
5         a)
6   state
    nfa.delta
```

Note: This move is different from the project. This move takes in an NFA, a **single** state, a symbol, and then returns all the states you can move to on that symbol. You can also assume no duplicate transitions show up and that this will not include epsilon as an input.

There are at least 3 bugs. For 3 of them, write down the line number and rewrite the **entire** line to fix it.

Line: _____ Fix: _____

Line: _____ Fix: _____

Line: _____ Fix: _____

Problem 3: Regex

[Total 10 pts]

(a) Equivalence [4 pts]

Indicate if the following regular expressions are equivalent

Regex 1	Regex 2	Yes	No
`a|b|c`	`[abc]`	(Y)	(N)
`aa+((b|c)?)*`	`a*[bc]*`	(Y)	(N)
`(c[a-zA-Z]+|c)`	`c[a-zA-Z]*`	(Y)	(N)
`[15-17]`	`15|16|17`	(Y)	(N)

(b) Write a Regex [6 pts]

Write a regular expression that would describe any valid, non-empty, lambda calculus string that has no parentheses. You can assume all variables are single lowercase characters. Whitespace does not matter.

Valid	Invalid
x	..a
λ x. a b x	$(\lambda x.x)$
$\lambda x.\lambda b.a$	$\lambda\lambda a.$

Problem 4: Lambda Calculus

[Total 10 pts]

(a) Give an expression that has no beta normal form under eager evaluation but does under lazy. (Hint: beta normal form means the term is fully evaluated with respect to beta-reduction.) [2 pts]

(b) Is $(z(\lambda x.xx)(\lambda x.xx))$ in Beta Normal Form? (Y) yes (N) no [2 pts]

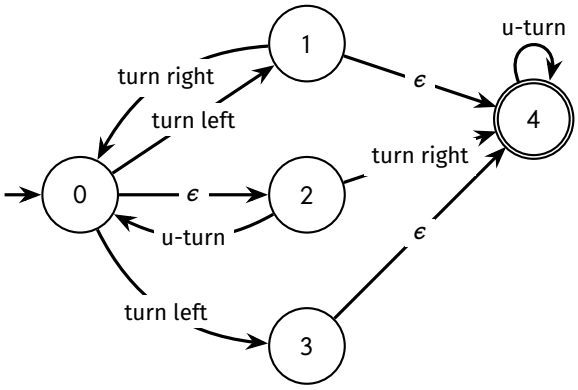
(c) Reduce the following to beta normal form. Show every step for full credit [6 pts]

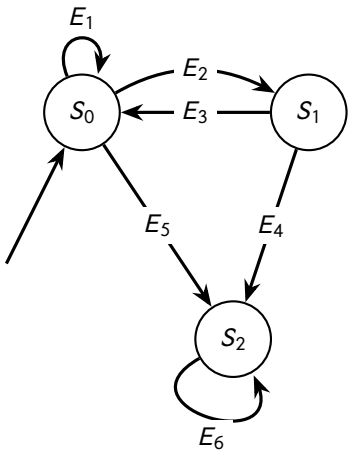
$$(\lambda x.\lambda z.z(xz)x)(\lambda a.za)(\lambda x.\lambda a.x)$$

Problem 5: FSM

[Total 8 pts]

(a) Convert the NFA **on the left** to the DFA **on the right**. You **MUST** show your work to get any credit. (You will need to remove the Garbage state) [6 pts]





Final?	State	turn left	turn right	u-turn

scratch Space:

S_0: S_1: S_2:

E_1: E_2: E_3:

E_4: E_5: E_6:

(b) Which states are the final (accepting) states? Select all that apply [2 pts]

(A) State S_0 (B) State S_1 (C) State S_2

Problem 6: CFGs

[Total 6 pts]

(a) derivation [4 pts]

Consider the following Grammar:
$S \rightarrow bSc \mid AS \mid c \mid \epsilon$
$A \rightarrow bA \mid \epsilon$

Derive the following string using a left-most derivation (do not draw a tree): `bbbcc`. You must show every step.

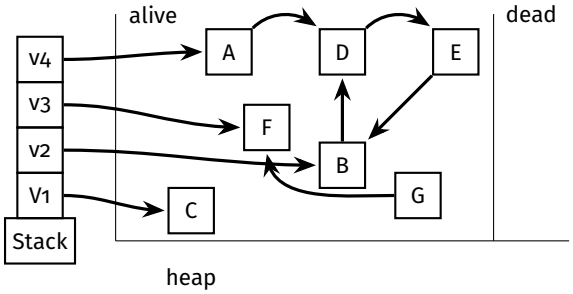
(b) Ambiguity [2 pts]

Is the above grammar ambiguous? (Y) yes (N) no

Problem 7: Garbage Collection

[Total 8 pts]

Given the following memory diagram:



Draw the stack and the alive portion of the heap after $v4$ has been popped off and then **stop and copy** was called. You do not have to draw the dead space, and for convenience we don't show it.

Problem 8: Type Checking

[Total 8 pts]

Consider the following Typing Rules for Ocaml:

$$\frac{}{G \vdash \text{true} : \text{bool}} \qquad \frac{}{G \vdash \text{false} : \text{bool}} \qquad \frac{}{G \vdash n : \text{int}}$$

$$\frac{G(x) = t}{G \vdash x : t} \qquad \frac{G \vdash e_1 : int \qquad G \vdash e_2 : int \qquad + = (int, int, int)}{G \vdash e_1 + e_2 : int}$$

$$\frac{G \vdash e_1 : t_1 \qquad G, x : t_1 \vdash e_2 : t_2}{G \vdash let\ x = e_1 \text{ in } e_2 : t_2} \qquad \frac{G \vdash e_1 : bool \qquad G \vdash e_2 : t \qquad G \vdash e_3 : t}{G \vdash if\ e_1\ then\ e_2\ else\ e_3 : t}$$

(a) Using the above rules, does `let x = if true then 3 else false in x + 5` typecheck? (Y) yes (N) no [2 pts]

(b) Write a type-checking proof for the expression [6 pts]

```
if true then let x = true in x else false
```

Problem 9: Rust

[Total 8 pts]

```
1 fn main(){
2   let mut x = String::from("hello");
3   let y = &mut x;
4   println!("{},{}",x,y);
5 }
```

Does the code compile? (Y) Yes (N) No

If **no**, explain why not in one sentence:

```
1 fn main(){
2   let x = String::from("Hello");
3   let mut y = x;
4   y.push_str(" world");
5   println!("{}",y);
6 }
```

Does the code compile? (Y) Yes (N) No

If **no**, explain why not in one sentence:

```
1 fn main(){
2   let mut x = String::from("Hello");
3   let mut y = String::from("World");
4   let mut a = &mut x;
5   a = &mut y;
6   println!("{}",a);
7 }
```

Does the code compile? (Y) Yes (N) No

If **no**, explain why not in one sentence:

```
1  fn function(s1: String,
2              s2: String,
3              f:bool)->usize{
4      if f {s1.len()} else{s2.len()}
5  }

6  fn main(){
7      let a = String::from("hello");
8      let b = a.clone();
9      let c = function(b,a,true);
10     println!("{} has length {}",a,c);
11 }
```

Does the code compile? (Y) Yes (N) No

If **no**, explain why not in one sentence:

Problem 10: Coding

[Total 20 pts]

(a) Ocaml [10 pts]

Many functions we have you write start with an initial value and works towards an accumulated value. Suppose we have a function that does the opposite: given an accumulated ending value, it produces the previous initial value.

Write a function called `make_history` that takes in such a function, an accumulated value, and an initial value. Its goal is to construct the list of accumulated values showing how we got from the initial value to the accumulated value.
See in the first example below: the function is applied to 3, the accumulated value, until it reaches 0, the initial value. The resulting list shows how we got from 0 to 3, inclusive of the accumulated and initial values.

You may not use imperative Ocaml. You can define recursive helper functions. Helpers can be defined below the function definition we give. You are not allowed to use any List module functions except those already given to you in the cheat sheet. Otherwise you may use anything in StdLib. You can assume that the accumulated and initial values are the same type. You can assume that you can get from the accumulated value to the initial value.

```
examples:
make_history (fun x -> x - 1) 3 0 = [0;1;2;3];;
make_history (fun x -> x ) true true = [true;true];;
make_history (fun lst -> match lst with [] -> [] |_::xs -> xs) [1;2;3] [] =
    [[];[3];[2;3];[1;2;3]];;

let rec make_history f acc initial =
```

(b) Rust [10 pts]

Do the same thing but in Rust. You can however assume that the input function will take in an `i32` and return an `i32`. Additionally, `acc` and `initial` will also be `i32`. Consequently, the return value for the function will be `Vec<i32>`.
You cannot use the unsafe keyword
Note: `Vec::push` will add to the end of the vector - the reference sheet may be helpful.

```
example:
fn prev_value(acc:i32)->i32{
    acc - 1
}

fn iden(acc:i32)->i32{
    acc
}

make_history(prev_value, 3,0) => [0,1,2,3]
make_history(iden, 3,3) => [3,3]


fn make_history(f: impl Fn(i32)->i32, acc:i32, initial:i32)->Vec<i32>{









}
```

Problem 11: Extra Credit [Total 2 pts]

For the following, you may only receive up to 2 points regardless of how many answers you give.

(a) Staff Stalking [1 pts]

What is your discussion TA's name and what is your discussion's section number?

(b) Staff Stalking [1 pts]

What is your discussion TA's eye color?

(c) Colon Parenthesis [1 pts]

Draw your discussion TA. You should include one identifiable feature so it's not just a generic stick figure. Pictures are up for interpretation. This will be graded by your discussion TA.

For Scratch Work - Do not tear off. Indicate the problem with your work

Reference Sheet

OCaml

```
(* Map and Fold *)
('a -> 'b) -> 'a list -> 'b list
let rec map f l = match l with
   [] -> []
  |x::xs -> (f x)::(map f xs)

('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
let rec fold_left f a l = match l with
   [] -> a
  |x::xs -> fold_left f (f a x) xs

('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
let rec fold_right f l a = match l with
   [] -> a
  |x::xs -> f x (fold_right f xs a)
```

Structure of Regex

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

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')
[abc]	match any character in abc
[^r_1]	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
(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

Rust

```
// Vectors
let vec = Vec::new();  // makes a new vector
let mut vec = vec![1,2,3]

vec.push(ele);  // Pushes the element 'ele'
                // to end of the vector 'vec'
vec.reverse();  // reverses the vector 'vec'
                // in place
```

Subset Construction Algorithm

NFA (input): $(\Sigma, Q, q_0, F_n, \delta)$, DFA (output): $(\Sigma, R, r_0, F_d, \delta_n)$

$R \leftarrow \{\}$
$r_0 \leftarrow \epsilon - \text{closure}(\sigma, q_0)$
while $\exists$ an unmarked state $r \in R$ **do**
  mark r
  for all $a \in \Sigma$ **do**
    $E \leftarrow \text{move}(\sigma, r, a)$
    $e \leftarrow \epsilon - \text{closure}(\sigma, E)$
    if $e \notin R$ **then**
      $R \leftarrow R \cup \{e\}$
    end if
    $\sigma_n \leftarrow \sigma_n \cup \{r, a, e\}$
  end for
end while
$F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$