CMSC330 - Organization of Programming Languages
Fall 2025 - Exam 1

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
- Please remove the reference sheet from the exam
- 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
P1.	10
P2.	6
P3.	5
P4.	10
P5.	12
P6.	14
P7.	13
P8.	9
P9.	5
P10.	16
Total	100

Problem 1: Concepts

[Total 10 pts]

The following questions include True/False and Multiple Choice formats.

In OCaml, functions are first-class constructs, meaning they can be passed as arguments, returned from functions, and stored in data structures. (T) F

Tail recursion in OCaml is important because it prevents stack overflow in recursive function calls. (T) F

With immutable state, there is less chance of data races. (T) F

Functional programming encourages mutable state and shared variables. T (F)

Map, filter, and fold are common higher-order functions used in functional programming. (T) F

For each of the following, select 1:

Currying in functional programming means:
(A) Converting a function with multiple arguments into a sequence of functions each taking one argument.
B Combining two functions into a single function.
C Using recursion instead of loops.
D Wrapping data into a function.

Which of the following must be an example of a higher-order function?
A A function that adds two integers.
B A function that calls itself recursively.
(C) A function that takes another function as input.
D A function that contains a loop.

Which of the following best describes the difference between map and fold?
(A) map creates a new list where each element corresponds to one from the input list, while fold combines elements into a single value.
B map reduces the input list to a single value, while fold creates a new list where each element corresponds to one from the original.
C Both perform the same operation.
D fold is only used for numerical operations.

A closure is:
(A) A function bundled together with its referencing environment.
B A recursive function that calls itself.
C A function that takes another function as input.
D A global function with no arguments.

What does the OCaml compiler do if you omit a case in a pattern match:
A Nothing
(B) Issues a warning
C Produces an error

Problem 2: Project 2

[Total 6 pts]

Below is Project 2's `tree` type and an implementation of `tree_fold`

```
type 'a tree = Node of 'a tree * 'a * 'a tree | Leaf

let rec tree_fold f init tree = match tree with
    | Leaf -> init
    | Node(l, v, r) -> f (tree_fold f init l) v (tree_fold f init r)
```

Use the `tree_fold` function to write the function `sum` of type:
(int tree → int)
that will return the sum of all the integer values in the tree.
You may not use recursion outside of `tree_fold`, meaning you may not add the `rec` keyword to `sum` nor any added helper.
Examples:

```
let t1 = Node(Node(Leaf, 1, Leaf), 2, Node(Leaf, 3, Leaf))
let t2 = Node(Node(Leaf, 1, Leaf), 2, Node(Node(Leaf, 3, Leaf), 4, Leaf))
sum t1 = 6
sum t2 = 10
```

```
let sum tree = tree_fold (fun l v r → l + v + r) 0 tree
```

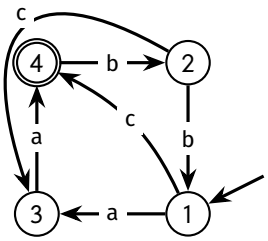
Problem 3: Finite State Machine

[Total 5 pts]

The following questions are independent from each other.

(a) Acceptance [5 pts]

Given the following Finite State Machine, select all of the strings it accepts:



(A) "" (empty string)
(B) ac
(C) aabbca
(D) abba
(E) aa
(F) cbb

Problem 4: Regex

[Total 10 pts]

(a) Digits [2 pts]

Which of the following regular expressions correctly matches any nonempty string consisting only of digits?

(A) `^[0-9]$`
(B) `^[0-9]+$`
(C) `^[0-9]*$`
(D) `^.*[0-9].*$`

(b) Zip Code [2 pts]

Which of the following regular expressions correctly matches a valid 5-digit ZIP code (e.g., 12345)?

(A) `^[0-9]{5}$`
(B) `^[0-9]{4,6}$`
(C) `^[a-zA-Z0-9]{5}$`
(D) `^ \d{5,}$`

(c) Time [6 pts]

Write a regular expression that exactly matches time strings in the 12-hour format (hh:mm:ss) with an AM/PM suffix.

- Hours range from 00 to 12.
- Minutes and seconds range from 00 to 59.
- The suffix can be any of am, pm, AM, or PM.
- No spaces are allowed in valid strings.

valid strings	**invalid strings**
`00:11:08am`	`12 AM`
`06:35:59pm`	`15:00:04pm`
`05:41:37AM`	`12:30PM`
`08:29:21PM`	`03:08:25`

(0[0-9]|1[0-2]):([0-5][0-9]):([0-5][0-9])(am|pm|AM|PM)

Problem 5: Debugging [Total 12 pts]

(a) Debug [4 pts]

John is trying to implement a function "all" that takes a list of booleans and returns true only if every element in the list is true, and false otherwise. However, his current implementation is not working correctly. Can you help him debug and fix the code?

You must write a corrected version of this code, not explain what is incorrect in words. There may be more than one issue to be resolved.

```
let all = List.fold_left (fun acc c-> acc || c) false;;
```

Fix:

```
List.fold_left (fun acc c → acc && c) true
```

(b) Fill in the blank [8 pts]

John is determined to implement a tail-recursive list reversal function, but parts of his code are missing. Can you help him complete it?

```
let rev l =
  let rec aux xs acc=
   match xs with
    |[] -> acc
    |h :: t -> aux t (h :: acc)

in aux l []
```

Problem 6: OCaml Typing [Total 14 pts]

Give the type of the function `'foo'`. If there is a type error, put "ERROR"

```
let foo a = if a then None else Some a;;
```

```
bool → bool option
```

```
let foo a b = [a;b]
```

```
'a → 'a → 'a list
```

```
let foo x f =
  match x with
  | h :: [] -> f (h :: [ "a" ])
  | h :: t -> f t
  | _ -> false
```

```
string list → (string list → bool) → bool
```

```
let foo a b c d =
    if (a b) then (c, d) else (b+1, (a c))
```

```
(int → bool) → int → int → bool → (int * bool)
```

Write an expression that has the following type, without using type annotations. All pattern matching must be exhaustive.

`int -> int list`

```
fun a → [a + 1]
```

`(string * 'a) -> string`

```
(fun a b) → a ^ "hello"
```

Problem 7: Evaluation

[Total 13 pts]

Evaluate each of the following OCaml expressions. If an expression causes a compilation error, write "ERROR". Use the `map` and `fold_left` implementations provided in the cheatsheet.

```
let foo xs =
    fold_left (fun a x -> x::a) [] xs in
foo (map (fun x -> x * 2) [3;6;9;12])
```

```
[24;18;12;6]
```

```
let foo =
    let x = ref 0 in
    fun a -> x := !x + a; !x in
map foo [1;2;3;4]
```

```
[1;3;6;10] ([10;9;7;4] is also accepted)
```

```
let foo a b = if a > b then a else b in
map (foo 1) [0;1;2;3]
```

```
[1;1;2;3]
```

```
let f x =
  if x = 90 then "A+"
  else 0 in
f 5
```

```
ERROR
```

```
type person = { name: string; age: int };;
let alice = { name = "Alice"; age = 30 };;
alice.name;;
```

```
"Alice"
```

Problem 8: Property Based Testing

[Total 9 pts]

In OCaml, a composed function combines two functions so that the output of one becomes the input of the next. For example:

```
let square x = x * x;;
let double x = 2 * x;;
let compose  x = square (double x);;
compose 2 = square (double 2) = 16
map compose [1;2;3] =  [4; 16; 36]
```

Consider the following INCORRECT `map` function for a list.

```
let rec map f lst = match lst with
   [] -> []
  |x::xs -> map f xs @ [f x]
```

Consider the following property *p* about the `map` function:
p: Mapping a composed function "let compose x = foo (bar x)" is equivalent to first mapping the function bar, and then mapping the function foo over the results.

Using a **correct** implementation of `map`, this property *p* should hold true for all valid inputs?

(●) Yes ( ) No

Using **our** implementation (shown above) of `map`, this property *p* should hold true for all valid inputs?

( ) Yes (●) No

Suppose I encode this property in OCaml to be used in OCaml's QCheck library as the following:

```
let prop f g xs = map (fun x -> f (g x)) xs = (map f (map g xs))
```

The above `prop` function is a valid encoding of the property *p*.

(●) Yes ( ) No

Problem 9: Error Handling

[Total 5 pts]

Match the following error messages with the example that can cause the error. Each example must only be used **once**, so choose the fix that fits the best.

Error	Answer
`Error: Syntax error`	D
`Warning 8 [partial-match]: this pattern-matching is not exhaustive.`	F
`Warning 11 [redundant-case]: this match case is unused.`	C
`Error: The value x has type int but an expression was expected of type int list`	E
`Exception: Division_by_zero`	A
`Exception`	B

	Example
A	1/0
B	List.hd []
C	let g x = match x with |*c* ⟶ "*one*" |1 ⟶ "*two*";;
E	fun x ⟶ (*x* :: [1], [1]@*x*)
D	let x = 1 +
F	let f x = match x with |1 ⟶ "*one*"

Problem 10: Coding

[Total 16 pts]

Restrictions for all coding questions: You are not allowed to use imperative OCaml; you can define recursive helper/helper functions below the function signatures we provided. You are not allowed to use any List module functions except the ones already given to you on the cheat sheet.

(a) Repeat [8 pts]

Define a function repeat : "'a ⟶ int ⟶ 'a list" that takes a value x and a non-negative integer n and returns a list containing n copies of x. Examples:

```
repeat 1 0 = [];;
repeat "hello" 2= ["hello"; "hello"];;*)
```

```
let repeat x n =
    let rec helper acc n =
        if n > 0 then
            helper (x :: acc) (n-1)
        else
            acc
    in helper [] n
```

(b) Expand [8 pts]

Define a function expand l : ('a * int) list ⟶ 'a list that takes a list of of pairs and returns a list of the first elements of those pairs, where those elements appear as many times as indicated by the second elements. (Hint: You may want to use the repeat function from the previous question)

```
expand [(2,1);(1, 3);(5,2)] = [2;1;1;1;5;5];;
expand [("hello",3);("four",0)] = ["hello";"hello";"hello"];;
```

```
let expand l =
    let rec helper acc l = match l with
        | [] -> acc
        | (a,b) :: t -> helper (acc @ (repeat a b)) t
    in helper [] l
```

Cheat 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

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

```
(* OCaml Function Types *)
:: -: 'a -> 'a list -> 'a list

@ -: 'a list -> 'a list -> 'a list

+, -, *, /  -: int -> int -> int
+., -., *., /. -: float -> float -> float

&&, || -: bool -> bool -> bool
not -: bool -> bool

^ -: string -> string

=>,>,=,<,<=, <> :- 'a -> 'a -> bool

(* Imperative OCaml *)
(* Example *)
let d = ref 0;;
val d : int ref = {contents = 0}
d := 1;;
- : unit = ()
!d;;
- : int = 1

(* Types *)

( ref ) : 'a -> 'a ref

( := );; - : 'a ref -> 'a -> unit = <fun>

( ! );; - : 'a ref -> 'a = <fun>
```

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')
[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