CMSC330 - Organization of Programming Languages
Fall 2024 - 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
- The back of the reference sheet has some scratch space on it. If you use it, you must turn in your scratch work
- 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.	15
P3.	10
P4.	25
P5.	20
P6.	20
Total	100

Problem 1: Language Concepts

[Total 10 pts]

	True	False
While a Regex cannot describe arbitrarily sized strings that are palindromes, FSMs can describe these strings If regex can't do it, FSMs cannot do it either	○	●
While a Regex can describe arbitrarily sized strings that are palindromes, FSMs cannot describe these strings If regex can't do it, FSMs cannot do it either	○	●
When converting an NFA of n states to a DFA using the subset construction method, the DFA will have m states where $n \leq m \leq 2^n$ You could have less states in the DFA than the NFA	○	●
When converting an NFA of n states to a DFA using the subset construction method, the DFA will have m states where $1 \leq m \leq 2^n$ Powerset would calculate all possible subset combinations	●	○
Order of evaluation matters in OCaml when you have to reason about mutable data constructs like references Function argument evaluation is undefined so references make it difficult to reason about	●	○
You can reverse a list with `map` (found on reference sheet) map cannot store what was previously seen unlike fold	○	●
You can reverse a list with `fold_left` (found on reference sheet) `fold_left (fun a x -> x::a) [] lst`	●	○
The type `'a -> int -> bool` is equivalent to the type `'a -> (int -> bool)` currying makes this equivalent	●	○
The type `'a -> int -> bool` is equivalent to the type `('a -> int) -> bool` The former is 2 inputs to bool, the latter is one input to bool	○	●
Property Based testing should be used in place of unit testing Use them together, not as a replacement for each other	○	●
`fold_left` is tail recursive The recursive call is in tail position	●	○
The OCaml expression `let x = x + 1` and the C statement `x++;` both update a pre-existing variable `x` OCaml doesn't update x, it makes a new binding (because in ocaml x is immutable)	○	●
OCaml is statically typed which means types are checked at compile time This is true	●	○
`fold_left` and `fold_right` will always return the same value given the same function, accumulator and list `fold_left (-) 0 [1;2;3] <> fold_right (-) [1;2;3] 0`	○	●
Every Regex can be converted to a FSM, but not every FSM can be converted to a Regex False, every fsm can be converted to Regex	○	●
Property Based testing can only be used for functional programming languages PBT can be used in any language	○	●
Currying translates a function that takes multiple arguments into a sequence of single argument functions. True by definition	●	○
Any function can be rewritten to be tail recursive Sometimes you need past calls to calculate the next function call	●	○
Tuples can be dynamically sized Tuples are fixed size once created	○	●
The result of calling `e-closure` on a non-empty list could be an empty list `e-closure` will always include the input state	○	●
`fold_left`'s second argument (the accumulator) cannot be a user defined type (like a record or variant) Could be anything as long as the types adhere to fold's type signature	○	●
The OCaml expression `let x = 4 in let x = x + 1` throws an error.	●	○

	True	False
Will make a binding of a variable *x* and the value 5		
Referential Transparency makes it easy to reason about your program	● T	○ F
Does make it easier to construct mathematical proofs about your program		
Ocaml uses type inference to determine a variable’s type	● T	○ F
This is true. You don’t have to explicitly state a type when making a variable binding		

Problem 2: OCaml Typing and Evaluation

[Total 15 pts]

For each of the following, give the type of the functions f and give what the function call evaluates to. **If there is a type error in the function**, put ”TYPE ERROR” for the type, and put ”ERROR” for the evaluation. If the function call does not follow the type of f, put ”ERROR” for the evaluation.

(a) [3 pts]

```
let f x y = map y x

f (fun x -> x + 1) [1;2;3];;
```

Type: ’a list -> (’a -> ’b) -> ’b list

Evaluation: ERROR

(b) [3 pts]

```
let f x y = map y x

f (fun x -> x * 5) [3;2;1];;
```

Type: ’a list -> (’a -> ’b) -> ’b list

Evaluation: ERROR

(c) [3 pts]

```
let foo (x:int list) y f =
    fold_left f y x

foo (fun x y -> y::x) [1;2;3] [];;
```

Type: int list -> ’a -> (’a -> int -> ’a) -> ’a

Evaluation: ERROR

(d) [5 pts]

```
let f x =
  fun y -> x:= y;!x;;

f (ref 3) 6;;
```

Type: ’a ref -> ’a -> ’a

Evaluation: 6

(e) [5 pts]

```
let f x =
  fun y -> x:= y;!x;;

f (ref 7) 2;;
```

Type: ’a ref -> ’a -> ’a

Evaluation: 2

(f) [5 pts]

```
let f x =
  fun y -> !x; x:= y;;

f (ref 3) 5;;
```

Type: ’a ref -> ’a -> unit

Evaluation: ()

(g) [7 pts]

```
let f x = match x with
   [] -> (fun x -> x - 1)
 |x::xs -> x;;

f [fun x -> 7];;
```

Type: `(int -> int) list -> int -> int`

Evaluation: `fun x -> 7`

(h) [7 pts]

```
let f x = match x with
   [] -> (fun x -> x + 9)
 |x::xs -> x;;

f [fun x -> 1];;
```

Type: `(int -> int) list -> int -> int`

Evaluation: `fun x -> 1`

(i) [7 pts]

```
let f x y = match x with
   [] -> (fun x -> x - 1)
 |x::xs -> x y;;

f [fun a b -> a + b] 5;;
```

Type: `('a -> int -> int) list -> 'a -> int -> int`

Evaluation: `fun b -> 5 + b`

Problem 3: Property Based Testing

[Total 10 pts]

Consider the `‘a tree` and `‘a flat` types from project 2:

```
type ’a tree =                                type ’a flat =
  | BiNode of ’a tree * ’a * ’a tree            | Lf
  | Leaf                                        | Nd of ’a
```

Consider an attempted (buggy!) implementation of the `flatten` function from project 2. `flatten` *should* convert a `’a tree` into a `’a flat list` using postorder (left, right, root) traversal:

```
let rec flatten input = match input with
  | Leaf -> []
  | BiNode (l, v, r) -> (Nd v) :: flatten l @ flatten r
```

Consider the property *p*: When flattening a `’a tree` of *n* BiNodes and *m* Leafs, you should get a `’a flat list` of length $n + m$

Is p a valid property? ● Yes ○ No

Does the current implementation of `flatten` maintain the property *p*? ○ Yes ● No

Suppose we wanted to write this test. We would encode the property as the following:

```
(* assume count_nodes returns the number of BiNodes and Leafs in a ’a tree and is correct *)
(* assume List.length returns the length of a list and is correct *)
let size_prop t = count_nodes t = List.length (flatten t)
```

Is `size_prop` a correct encoding of the property p? ● Yes ○ No

Consider an attempted (buggy!) implementation of the `flatten` function from project 2. `flatten` *should* convert a `’a tree` into a `’a flat list` using postorder (left, right, root) traversal:

```
let rec flatten input = match input with
  | Leaf -> [Lf]
  | BiNode (l, v, r) -> (Nd v) :: flatten l @ flatten r
```

Consider the property *p*: When flattening a `’a tree` of *n* BiNodes and *m* Leafs, you should get a `’a flat list` of length $n + m$

Is p a valid property? ● Yes ○ No

Does the current implementation of `flatten` maintain the property *p*? ● Yes ○ No

Suppose we wanted to write this test. We would encode the property as the following:

```
(* assume count_nodes returns the number of BiNodes and Leafs in a ’a tree and is correct *)
(* assume List.length returns the length of a list and is correct *)
let size_prop t = count_nodes t = List.length (flatten t)
```

Is `size_prop` a correct encoding of the property p? ● Yes ○ No

Consider an attempted (buggy!) implementation of the `flatten` function from project 2. `flatten` *should* convert a binary tree into a `’a flat list` using postorder traversal:

```
let rec flatten input =
  match input with
  | Leaf -> [ Lf ]
  | BiNode (l, v, r) -> (Nd v) :: flatten l @ flatten r
```

Consider the property *p*: When flattening a `’a tree` *t*, the root of *t* should be the last item of the resulting `’a flat list`

Is p a valid property? [✓ Yes] No

Does the current implementation of `flatten` maintain the property *p*? Yes [✓ No]

Suppose we wanted to write this test. We would encode the property as the following:

```
let root_prop t =
    let rec last l = match l with
       [] -> raise (Failure "should have something")
      |[x] -> x
      |x::xs -> last xs in
    match (t,last (flatten t)) with
      BiNode(y), Nd(x) -> y = x
     |Leaf, Lf -> true
     |_,_ -> false
```

Is `root_prop` a correct encoding of the property p? Yes [✓ No]

Problem 4: Coding and Debugging

[Total 25 pts]

(a) Given an `int list list`, write a function called `product_sum` that sum up the product of every row. You are allowed to write recursive helper functions, but you may not use imperative OCaml. You also cannot use any List module functions that are not found on the reference sheet. [10 pts]

```
mtx = [ [1;2;3;4;5;6]; (* 1 * 2 * 3 * 4 * 5 * 6 = 720 *)
        [6;5;4;3;2;1]; (* 720 *)
        [1;1;1;1;1;1]; (* 1 *)
        [9;8;7;6;5;4]  (* 60480 *)
      ]
(* 720 + 720 + 1 + 60480 = 61921 *)
product_sum mtx = 61921

let product_sum mtx = fold_left (fun a x -> a + fold_left (*) 1 x) 0 mtx
```

(b) Given an `int list list`, write a function called `sum_diff` that subtracts the sum of every row starting at 0. You are allowed to write recursive helper functions, but you may not use imperative OCaml. You also cannot use any List module functions that are not found on the reference sheet. [10 pts]

```
mtx = [ [1;2;3;4;5;6]; (* 1 + 2 + 3 + 4 + 5 + 6 = 21 *)
        [6;5;4;3;2;1]; (* 21 *)
        [1;1;1;1;1;1]; (* 6 *)
        [9;8;7;6;5;4]  (* 39 *)
      ]
(* 0 - 21 - 21 - 6 - 39 = -87 *)
sum_diff mtx = -87

let sum_diff mtx = fold_left (fun a x -> a - fold_left (+) 1 x) 0 mtx
```

(c) Write a function called `get` which takes indices i, j, and a matrix in `int list list` format, and returns the value at row i and column j. Indices start at 0. If the i or j index is out range, throw an exception using `Failwith "out of range"`. You may write helper functions, but you may not use imperative OCaml [10 pts]

```
mtx = [ [1;2;3;4;5;6];
        [6;5;4;3;2;1];
        [1;1;1;1;1;1];
        [9;8;7;6;5;4]
      ]
get mtx 3 4 = 5
get mtx 0 0 = 1

get mtx  4 0;;
Exception: Failure "out of range".

get mtx  1 6;;
Exception: Failure "out of range".

let get mtx i j =
  let rec index l x = match l with
    [] -> raise (Failure "out of range")
    |n::ns -> if x = 0 then n else index ns (x - 1) in
  index (index mtx i) j
(* could also be: let get mtx i j = List.nth i (List.nth j mtx) *)
```

Recall the `'a n_tree` from project 2 defined below. Debug the following code used to add 5 to each node in an `int n_tree`. There are two (2) type errors and one (1) logic bug. For the logic bug, we provide an input that returns the incorrect value. Things that would cause warnings are not bugs in this case. A **Type Error** is one which the OCaml Type Checker will catch. A **logic bug** is one which the Ocaml type checker will not warn you about. The **fix** should be a segment of code to replace all or part of the line you indicate.

```
type 'a n_tree = Node of 'a * 'a n_tree list

1 let rec tree_map f t = match t with
2     Node(v,[]) -> Leaf
3    |Node(x,lst) -> Node(x,List.map (tree_map f) lst)

4 let add5 t = tree_map t (fun x -> x + 5)

(* add5 Node(0,[Node(1,[Node(3,[])]);Node(2,[])]) gives incorrect output *)
```

(d) **Type Error 1** [5 pts]

Line: 2 Fix: `-> Node(f v, [])`

(e) **Type Error 2** [5 pts]

Line: 4 Fix: `tree_map (fun x -> x + 5) t`

(f) **Logic Bug** [5 pts]

Line: 3 Fix: `-> Node(f x, ...)`

Recall the n_tree from project 2 defined below. Debug the following code used to trim an n_tree to height *n* (a tree with no children is considered to have height 0). There are two (2) type errors and one (1) logic bug. For the logic bug, we provide an input that returns the incorrect value. Things that would cause warnings are not bugs in this case. A **Type Error** is one which the OCaml Type Checker will catch. A **logic bug** is one which the Ocaml type checker will not warn you about. The **fix** should be a segment of code to replace all or part of the line you indicate.

```
type 'a n_tree = Node of 'a * 'a n_tree list

1 let rec trim n t = match t with
2     Node (v,[]) -> Node(t,[])
3    |Node(v,x::xs) -> if n > 0 then Node(v, List.map (trim (n-1)) xs)
4                      else Node(x,[])

(* Trim 1 Node(1,[Node(3,[Node(4,[])])]) gives incorrect output *)
```

(g) **Type Error 1** [5 pts]

Line: 2 Fix: `-> Node(v,[])`

(h) **Type Error 2** [5 pts]

Line: 4 Fix: `else Node(v,[])`

(i) **Logic Bug** [5 pts]

Line: 3 Fix: `... List.map (trim (n-1)) (x::xs)`

Problem 5: Regex

[Total 20 pts]

(a) Assuming a full match, which strings are accepted by the following regex? Select all that apply. (Note: there is a single space after the : symbol in both the question and the answer choices) [6 pts]

```
^(Small|Large) number: -?[0-9]\.[0-9]x10\^(-?[0-9])$
```

○ A `Small number: -4.1x10^23` ○ B `large number: 5.0^4` ○ C `Small number: 50`

○ D `2.3x10^-4` ● E `Small number: 0.0x10^-0` ● F `Large number: -1.3x10^7`

(b) Assuming a full/exact match, which strings are accepted by the following regex? Select all that apply. (Note: there is a single space after the : symbol in both the question and the answer choices) [6 pts]

```
^(bin|hex|dec|oct) number: (0[obx])?([A-Z0-9]+|(0|1)*|[0-7]|-?[0-9]3)$
```

● A `bin number: 0b010101` ● B `oct number: 0oFFFFF` ○ C `dec number: -000F`

○ D `07644` ○ E `hex number: 0b-666` ○ F `bin number: 0obx`

(c) Are the following Regular expressions equivalent (assume exact match)? [6 pts]

Regex 1	Regex 2	Yes	No
`[a-f]+|[g-k]+`	`[a-k]+`	○ Y	● N
`[0-9]?[0-9]*[a-z]+`	`[0-9]*[a-z][a-z]*`	● Y	○ N
`b(a?b)*`	`(ba|b)*b*`	○ Y	● N
`[a-m]+|[n-z]+`	`[a-z][a-z]*`	○ Y	● N
`[0-9]?[0-9]*[a-z][a-z]*`	`[0-9]*[0-9]*[a-z]+`	● Y	○ N

(d) Let Bindings [8 pts]

Write a regex that describes strings that represent OCaml let bindings that bind variables to `int lists` with restrictions. EG: `let var = [1;2;3;]`. See the following restrictions:

- Variable names are any alphanumeric word of at least length 1 that does not begin with a capital or digit
- Ints can be positive and negative and a value like −0 is valid
- Ints can be 0-paddded so 02 is valid
- Ints can only be between -20 and 20 (inclusive)
- The list **can be empty**
- For simplicity, every element is followed by ;

```
let [a-z][A-Za-z0-9]* = \[(-?0*(([0-1]?[0-9])|20);)*\]
```

(e) Let Bindings [8 pts]

Write a regex that describes strings that represent OCaml let bindings that bind variables to `int lists` with restrictions. EG: `let hvar = [1;2;3;]`. See the following restrictions:

- Variable names are any alphanumeric word of at least length 1 that begin with the lowercase letter "h"
- Ints can be positive and negative and a value like −0 is valid
- Ints can be 0-paddded so 02 is valid
- Ints can only be between -20 and 20 (inclusive)
- The list **can be empty**
- For simplicity, every element is followed by ;

```
let h[A-Za-z0-9]* = \[(-?0*(([0-1]?[0-9])|20);)*\]
```

(f) Variants [8 pts]

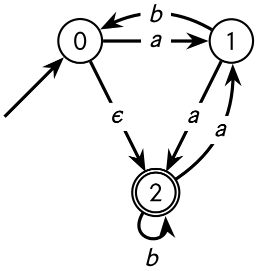
Write a regex that describes strings that represent OCaml Variant definitions with restrictions. EG:
type my_type = Tipe of int.
See the following restrictions:

- Variant type names (like `my_type`) are of length 3 or more that start with a lowercase character
- Variant type names can also contain Uppercase characters, digits, underscores (_), and dashes (-)
- Variant Types (like `Tipe`) start with a Capital followed by zero or more lowercase characters
- Variant can hold `ints` or `bools`, or tuples of size 2 or more that contain only `ints` or `bools`
- Variants cannot hold tuples of tuples

```
type [a-z][A-Za-z0-9_-]{2,} = [A-Z][a-z]* of (int|bool)(\*(int|bool))*
```

Problem 6: FSM

[Total 20 pts]

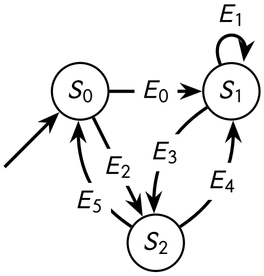


(a) What strings are accepted by this NFA? Select all that apply. [5 pts]

● A aaab ○ B abba ○ C a

● D baa ● E ϵ (Empty string) ● F bbbb

(b) Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state) [12 pts]

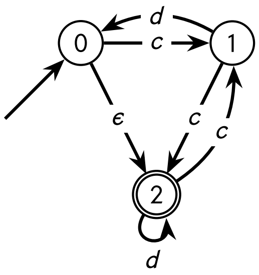


Work Space:

S_0:	0,2	S_1:	2	S_2:	1
E_0:	b	E_1:	b	E_2:	a
E_3:	a	E_4:	a	E_5:	b

(c) Which states are the final (accepting) states? Select all that apply [3 pts]

● A State S_0 ● B State S_1 ○ C State S_2

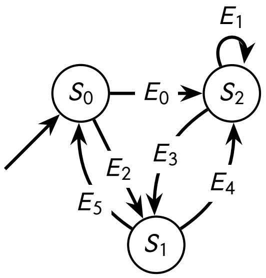


(d) What strings are accepted by this NFA? Select all that apply. [5 pts]

[✓ A] ϵ (Empty string) B cddc [✓ C] dddd

[✓ D] dcc [✓ E] cccd F c

(e) Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state) [12 pts]

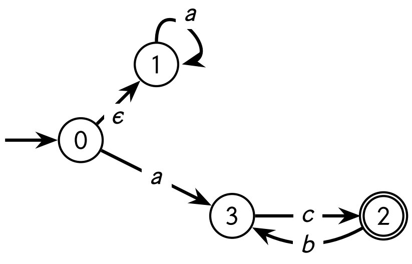


Work Space:

S_0:	0,2	S_1:	1	S_2:	2
E_0:	d	E_1:	d	E_2:	c
E_3:	c	E_4:	c	E_5:	d

(f) Which states are the final (accepting) states? Select all that apply [3 pts]

[✓ A] State S_0 B State S_1 [✓ C] State S_2



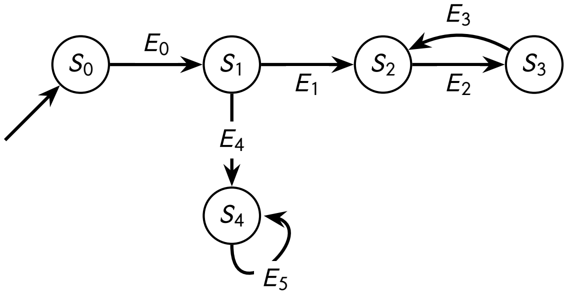
(g) What strings are accepted by this NFA? Select all that apply. [5 pts]

[✓ A] ac (B) aaac (C) ab

(D) cbc (E) ϵ (Empty string) [✓ F] acbc

(h) Convert the above NFA to the below DFA. You **MUST** show your work to get any credit [12 pts]

Work Space:



S_0: 0,1 S_1: 1,3 S_2: 2

S_3: 3 S_4: 1

E_0: a E_1: c E_2: b

E_3: c E_4: a E_5: a

(i) Which states are the final (accepting) states? Select all that apply [3 pts]

(A) State S_0 (B) State S_1 [✓ C] State S_2 (C) State S_3 (C) State S_4

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)



(* 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

ˆ  -: string -> string -> string

=>,>,=,<,<=, <> :-  ’a -> ’a -> bool
```

```
(* Regex in OCaml *)
Re.Posix.re: string -> regex
Re.compile: regex -> compiled_regex

Re.exec: compiled_regex -> string -> group
Re.execp: compiled_regex -> string -> bool
Re.exec_opt: compiled_regex -> string -> group option

Re.matches: compiled_regex -> string -> string list

Re.Group.get: group -> int -> string
Re.Group.get_opt: group -> int -> string option
```

Structure of Regex

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

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
$r_1 \mid r_2$	r_1 or r_2 (eg. a|b means ’a’ or ’b’)
[abc]	match any character in abc
[ˆr_1]	anything except r_1 (eg. [ˆabc] 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
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

NFA to DFA Algorithm (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
$F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$
```