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
(a) Every Regex can be converted to a FSM, but not every FSM can be converted to a Regex	T	F
(b) Property Based testing can only be used for functional programming languages	T	F
(c) Currying translates a function that takes multiple arguments into a sequence of single argument functions.	T	F
(d) Any function can be rewritten to be tail recursive	T	F
(e) Tuples can be dynamically sized	T	F
(f) The result of calling `e-closure` on a non-empty list could be an empty list	T	F
(g) `fold_left`'s second argument (the accumulator) cannot be a user defined type (like a record or variant)	T	F
(h) The OCaml expression `let x = 4 in let x = x + 1` throws an error.	T	F
(i) Referential Transparency makes it easy to reason about your program	T	F
(j) Ocaml uses type inference to determine a variable's type	T	F

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

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

Type:

Evaluation:

(b) [5 pts]

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

f (ref 3) 5;;
```

Type:

Evaluation:

(c) [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:

Evaluation:

Problem 3: Coding and Debugging [Total 25 pts]

(a) 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 =
```

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 *)
```

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

Line: Fix:

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

Line: Fix:

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

Line: Fix:

Problem 4: 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 binary tree into a `'a flat list` using postorder traversal:

```
let rec flatten =
  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 5: Regex

[Total 20 pts]

(a) 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`

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

Regex 1	Regex 2	Yes	No
`[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)
`b(a?b)*`	`(ba|b)+b*`	(Y)	(N)

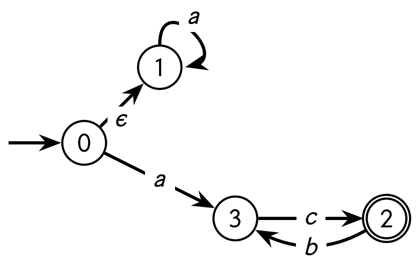
(c) Variants [8 pts]

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

Problem 6: FSM

[Total 20 pts]



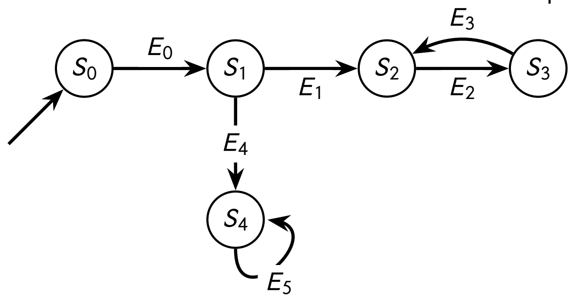
(a) What strings are accepted by this NFA? Select all that apply. [5 pts]

(A) ac (B) aaac (C) ab

(D) cbc (E) ϵ (Empty string) (F) acbc

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

Work Space:



S_0:		S_1:		S_2:	
S_3:		S_4:			
E_0:		E_1:		E_2:	
E_3:		E_4:		E_5:	

(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 (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

^  -: 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

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

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
$r_1 \vert r_2$	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**
$F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$
```