CMSC330 - Organization of Programming Languages
Spring 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	20
P2.	15
P3.	10+2
P4.	20
P5.	15
P6.	15
Total	95+2

Problem 1: Language Concepts

[Total 20 pts]

	True	False
Regular Expressions can match palindromic strings of arbitrary size	T	F
Every Regex can be converted into a FSM	T	F
Fold can be implemented in terms of Map	T	F
A function that checks acceptance on NFAs would not work on DFAs	T	F
Side effects can occur when using mutable data types	T	F
`fun x y -> y x` is an example of a higher order function	T	F
Functions are treated as data in OCaml	T	F
Checking acceptance of a DFA is O(n), where *n* is the length of the string	T	F
`let f x = x + 1` and `let f = fun x -> x + 1` are semantically the same	T	F
`let x = 1 in let x = 2 in x` is an example of variable shadowing in OCaml	T	F

Problem 2: OCaml Typing and Evaluation

[Total 15 pts]

Give the type for the following functions. Then give what the expression evaluates to. If there is an error in evaluation, put "ERROR". If there is an error in typing, put "ERROR" for both parts.

(a) [3 pts]

```
let f x y = match x with
   [] -> y
  |(a,b)::xs -> [a] @ [b];;

f [(1,2)] [5];;
```

Type:

Evaluation:

(b) [5 pts]

```
let f =
    let m = ref 0 in
    fun n -> m:=n; !m + n;;

(f 1) + (f 1) + (f 1);;
```

Type:

Evaluation:

(c) [7 pts]

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

f 3 (fun x -> 7);;
```

Type:

Evaluation:

Problem 3: Data Types with Fold and Map [Total 10+2 pts]

Consider the data type:

```
type 'a dir = Left of 'a * 'a dir | Right of 'a * 'a dir | END
```

(a) Write a function `dir_map` that behaves the same as `map` but will iterate over a `dir`. [5 pts]

```
dir_map (fun x -> x + 1) Left(0,Right(1,END)) -> Left(1,Right(2,END))
dir_map (fun x -> if x then 1 else 0) Left(true,Right(false,END)) -> Left(1,Right(0,END))

let rec dir_map f dirs =
```

(b) Write a function `num_line` that takes in an initial `int` value, and a `int dir`. The function returns the result of adding all `Right` values and subtracting `Left` values from the initial value. You may do this recursively or using the provided `fold_dir` function. This function acts like `fold_left` however it folds over `dir` types and modifies the value passed into the function. You may get up to 2 bonus points for using only `fold_dir` and non-recursive helper functions. [5+2 pts]

```
num_line 0 Right(1,Right(2,Left(3,END))) = 0
num_line 5 Right(-3,Left(2,END)) = 0
num_line 0 END = 0
num_line 5 END = 5

fold_dir: ('a -> 'b * bool -> 'a) -> 'a -> 'b dir -> 'a
let rec fold_dir f a d = match d with
   END -> a
  |Left(x,xs) -> fold_dir f (f a (x,true)) xs
  |Right(x,xs) -> fold_dir f (f a (x,false)) xs

let rec num_line init dirs =
```

Problem 4: Coding and Debugging [Total 20 pts]

(a) Recall part 2 of project 2. You were given the following tree and had to flatten it. Write a function cutie2list which converts a perfect Binary tree to a list using preorder (root, left, right) traversal, leaving out the LEAF. [8 pts]

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

(* example *)
cutie2list BiNode(BiNode(Leaf,2,Leaf),1,BiNode(Leaf,3,Leaf)) = [1;2;3]

let rec cutie2list tree =
```

Recall the other tree type from project 2, the `n_tree` defined below. Debug the following code used to mirror an `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.

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

1 let rec reverse lst = fold_left (fun a x -> a@[x]) [] lst

2 let rec mirror t = match t with
3    Node (v,[]) -> Leaf
4   |Node(x,lst) -> Node(x,reverse(map mirror [lst]))

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

(b) **Type Error 1** [4 pts]

Line: Fix:

(c) **Type Error 2** [4 pts]

Line: Fix:

(d) **Logic Bug** [4 pts]

Line: Fix:

Problem 5: Regex

[Total 15 pts]

(a) Assuming a full match, which strings are accepted by the following regex? Select all that apply [2 pts]

```
^let [a-z][a-zA-Z0-9]* = (-?[0-9]+);;$
```

(A) `let var1 = 3 + 4;;` (B) `let Variant = 4;;` (C) `let v = -03;;`

(D) `stmt = 0` (E) `let tweleve = twelve;;` (F) `let seven = --7;;`

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

Regex 1	Regex 2	Yes	No
`[a-d0-5]c*`	`(a|b|c|d|0|1|2|3|4|5)c*`	(Y)	(N)
`[aeiou]+a{2}`	`[aeiou][aeiou]*(aa)`	(Y)	(N)
`(a|b|c){3}d4,`	`abc|acb|[bac]3dddd*`	(Y)	(N)

(c) Dates [7 pts]

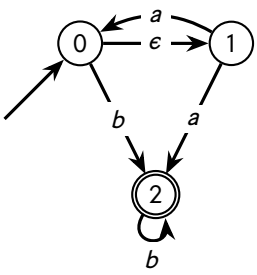
Write a regex that accounts for dates in `dd/mm/yyyy` format. Consider the following restrictions/modifications:

- Every month has 31 days.
- 00 is not a valid day nor month but 0000 is a valid year
- There are 12 months of the year and are 0 padded: 04 would be April

Problem 6: FSM

[Total 15 pts]

Scratch Space:



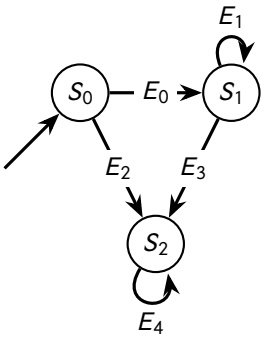
(a) What strings are accepted by this NFA? [2 pts]

(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 [10 pts]

Work Space:



S_0: S_1: S_2:

E_0: E_1: E_2:

E_3: E_4:

(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

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

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

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

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
$F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$