# CMSC330 - Organization of Programming Languages
# Spring 2024 - Exam 1 Solutions

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] |
| Regular Expressions cannot match palindromic strings of arbitrary size | [✓ T] | F |
| Every Regex can be converted into a FSM | [✓ T] | F |
| Not every Regex can be converted into a FSM | T | [✓ F] |
| Fold can be implemented in terms of Map | T | [✓ F] |
| Map can be implemented in terms of Fold | [✓ T] | F |
| A function that checks acceptance on NFAs would not work on DFAs | T | [✓ F] |
| A function that checks acceptance on NFAs would work on DFAs | [✓ T] | F |
| Side effects can occur when using mutable data types | [✓ T] | F |
| It is not possible for side effects to occur when using mutable data types | T | [✓ F] |
| `fun x y -> y x` is an example of a higher order function | [✓ T] | F |
| `fun y x -> y x` is an example of a higher order function | [✓ T] | F |
| `fun a b -> b a` is an example of a higher order function | [✓ T] | F |
| `fun a b -> a b` is an example of a higher order function | [✓ T] | F |
| Functions are treated as data in OCaml | [✓ T] | F |
| Functions are not 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 |
| Checking acceptance of a DFA is O($n^2$), 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 f x = x + 1` and `let f = fun x -> x + 1` are syntactically the same | T | [✓ F] |
| `let x = 1 in let x = 2 in x` is an example of variable shadowing in OCaml | [✓ T] | F |
| `let z = 5 in let x = 2 in z` is an example of variable shadowing in OCaml | T | [✓ F] |
| `let z = 1 in let z = 4 in z` is an example of variable shadowing in OCaml | [✓ T] | F |
| `let x = 4 in let z = 7 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:** `('a * 'a) list -> 'a list -> 'a list`

**Evaluation:** `[1; 2]`

(b) [5 pts]

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

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

**Type:** `int -> int`

**Evaluation:** `6`

(c) [7 pts]

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

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

**Type:** `'a -> ('a -> 'b) -> 'b`

**Evaluation:** `7`

## Problem 3: Data Types with Fold and Map

[Total 10+2 pts]

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

**Version A, C:**
Consider the data type:

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

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 = match dirs with
    |END -> END
    |Left(x,xs) -> Left(f x, dir_map f xs)
    |Right(x,xs) -> Right(f x, dir_map f xs)
```

**Version B, D:**
Consider the data type:

```
type 'a dir = Up of 'a * 'a dir | Down of 'a * 'a dir | END

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

```
let rec dir_map f dirs = match dirs with
    |END -> END
    |Up(x, xs) -> Up(f x, dir_map f xs)
    |Down(x, xs) -> Down(f x, dir_map f xs)
```

(b) Write num_line. [5+2 pts]

**Version A, C fold_dir:**

```
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
```

**Version A:**
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.

```
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
```

```
(* using fold_dir *)
let num_line init dirs =
    let f a (x, left) = if left then a + x else a - x in
    fold_dir f init dirs

(* not using fold_dir *)
let rec num_line init dirs =match dirs with
    |END -> init
    |Left(x, xs) -> num_line (init + x) xs
    |Right(x, xs) -> num_line (init - x) xs
```

**Version C:**
Write a function num_line that takes in an initial int value, and a int dir. The function returns the result of adding all Left values and subtracting Right values from the initial value.

```
num_line 0 Right(1,Right(2,Left(3,END))) = 0
num_line 5 Right(-3,Left(2,END)) = 10
num_line 0 END = 0
num_line 5 END = 5
```

```
(* using fold_dir *)
let num_line init dirs =
    let f a (x, left) = if left then a - x else a + x in
    fold_dir f init dirs

(* not using fold_dir *)
let rec num_line init dirs = match dirs with
    |END -> init
    |Left(x, xs) -> num_line (init - x) xs
    |Right(x, xs) -> num_line (init + x) xs
```

**Version B, D `fold_dir`:**

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

**Version 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 Down values and subtracting Up values from the initial value.

```
num_line 0 Down(1,Down(2,Up(3,END))) = 0
num_line 5 Down(-3,Up(2,END)) = 0
num_line 0 END = 0
num_line 5 END = 5
```

```
(* using fold_dir *)
let num_line init dirs =
    let f a (x, up) = if up then a - x else a + x in
    fold_dir f init dirs

(* not using fold_dir *)
let rec num_line init dirs = match dirs with
    |END -> init
    |Up(x, xs) -> num_line (init - x) xs
    |Down(x, xs) -> num_line (init + x) xs
```

**Version D:**
Write a function num_line that takes in an initial int value, and a int dir. The function returns the result of adding all Up values and subtracting Down values from the initial value.

```
num_line 0 Down(1,Down(2,Up(3,END))) = 0
num_line 5 Down(-3,Up(2,END)) = 10
num_line 0 END = 0
num_line 5 END = 5
```

```
(* using fold_dir *)
let num_line init dirs =
    let f a (x, up) = if up then a + x else a - x in
    fold_dir f init dirs

(* not using fold_dir *)
let rec num_line init dirs = match dirs with
    |Up(x, xs) -> num_line (init + x) xs
    |Down(x, xs) -> num_line (init - x) xs
```

## Problem 4: Coding and Debugging [Total 20 pts]

(a) `cutie2list` [8 pts]

**`tree` definition:**

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

**Version A, C:** Write a function cutie2list which converts a perfect Binary tree to a list using preorder (root, left, right) traversal, leaving out the LEAF.

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

```
let rec cutie2list tree = match tree with
    |Leaf -> []
    |BiNode(left, node, right) -> [node] @ (cutie2list left) @ (cutie2list right)
```

**Version B:** Write a function cutie2list which converts a perfect Binary tree to a list using postorder (left, right, root) traversal, leaving out the LEAF.

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

```
let rec cutie2list tree = match tree with
    |Leaf -> []
    |BiNode(left, node, right) -> (cutie2list left) @ (cutie2list right) @ [node]
```

**Version D:** Write a function cutie2list which converts a perfect Binary tree to a list using inorder (left, root, right) traversal, leaving out the LEAF.

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

```
let rec cutie2list tree = match tree with
    |Leaf -> []
    |BiNode(left, node, right) -> (cutie2list left) @ [node] @ (cutie2list right)
```

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: | 3 | Fix: | Leaf should be `Node(v, [])` as Leaf is not defined for `n_tree` |

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

| | | | |
|---|---|---|---|
| Line: | 4 | Fix: | `[lst]` should just be `lst` since it's already an `'a n_tree lst` type |

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

| | | | |
|---|---|---|---|
| Line: | 1 | Fix: | `a @ [x]` should be `[x] @ a` since `fold_left` works from left to right |

## 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]

**Version A:**

```
^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;;`

**Version B:**

```
^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;;`

**Version C:**

```
^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;;`]

**Version D:**

```
^(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]

**Version A, D:**

| 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] |

**Version B:**

| Regex 1 | Regex 2 | Yes | No |
|---|---|---|---|
| `[a-d0-5]c*` | `(a\|b\|c\|d\|0\|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] |

**Version C:**

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

(c) Dates [7 pts]

**All the versions use these restrictions:**

- 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

**Version A:**
Write a regex that accounts for dates in dd/mm/yyyy format.

```
(0[1-9]|1[0-9]|2[0-9]|3[0-1])\/(0[1-9]|1[0-2])\/[0-9]{4}
```

**Version B:**
Write a regex that accounts for dates in mm/dd/yyyy format.

```
(0[1-9]|1[0-2])\/(0[1-9]|1[0-9]|2[0-9]|3[0-1])\/[0-9]{4}
```

**Version C:**
Write a regex that accounts for dates in yyyy-mm-dd format.

```
[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|1[0-9]|2[0-9]|3[0-1])
```

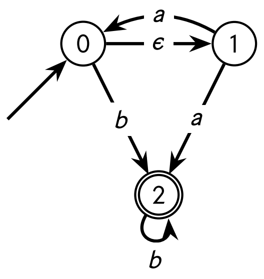
**Version D:**
Write a regex that accounts for dates in mm-dd-yyyy format.

```
(0[1-9]|1[0-2])-(0[1-9]|1[0-9]|2[0-9]|3[0-1])-[0-9]{4}
```

## Problem 6: FSM

[Total 15 pts]

**Version A:**

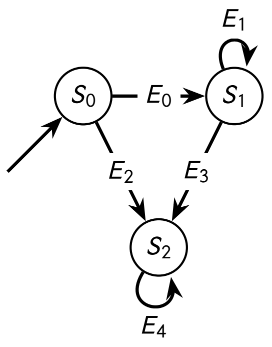


What strings are accepted by this NFA?

[✓ A] aaab   B abba   [✓ C] a

D baa   E $\epsilon$ (Empty string)   [✓ F] bbbb

Convert the above NFA to the below DFA.

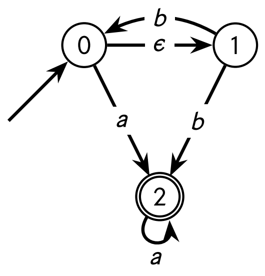


| | | | | | |
|---|---|---|---|---|---|
| $S_0$: | [0, 1] | $S_1$: | [0, 1, 2] | $S_2$: | [2] |
| $E_0$: | a | $E_1$: | a | $E_2$: | b |
| $E_3$: | b | $E_4$: | b | | |

Which states are the final (accepting) states? Select all that apply

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

**Version B:**

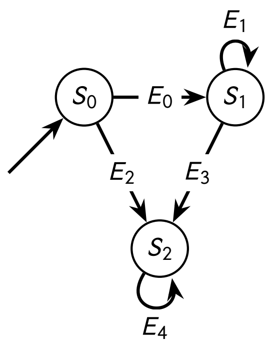


What strings are accepted by this NFA?

(A) aaab (B) abba [✓ C] a

[✓ D] baa (E) $\epsilon$ (Empty string) [✓ F] bbbb

Convert the above NFA to the below DFA.

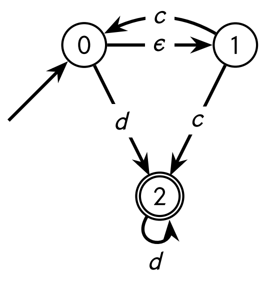


| | | | | | |
|---|---|---|---|---|---|
| $S_0$: | [0, 1] | $S_1$: | [0, 1, 2] | $S_2$: | [2] |
| $E_0$: | b | $E_1$: | b | $E_2$: | a |
| $E_3$: | a | $E_4$: | a | | |

Which states are the final (accepting) states? Select all that apply

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

**Version C:**

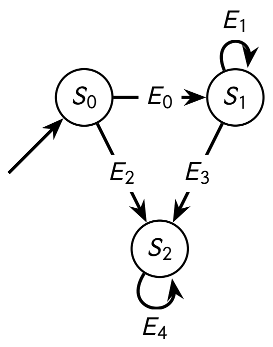


What strings are accepted by this NFA?

[✓ A] cccd (B) cddc [✓ C] c

[✓ D] cdd (E) $\epsilon$ (Empty string) [✓ F] d

Convert the above NFA to the below DFA.

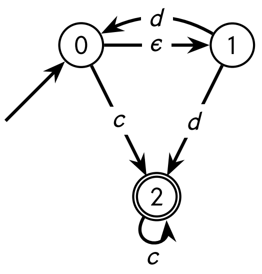


| | | | | | |
|---|---|---|---|---|---|
| $S_0$: | [0, 1] | $S_1$: | [0, 1, 2] | $S_2$: | [2] |
| $E_0$: | c | $E_1$: | c | $E_2$: | d |
| $E_3$: | d | $E_4$: | d | | |

Which states are the final (accepting) states? Select all that apply

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

**Version D:**

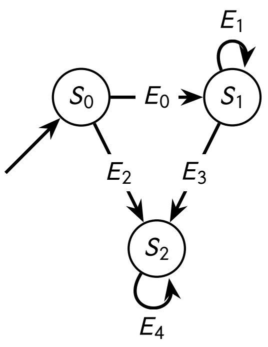


What strings are accepted by this NFA?

(A) cccd (B) cddc [✓ (C) c]

(D) cdd (E) $\epsilon$ (Empty string) [✓ (F) d]

Convert the above NFA to the below DFA.



| | | | | | |
|---|---|---|---|---|---|
| $S_0$: | [0, 1] | $S_1$: | [0, 1, 2] | $S_2$: | [2] |
| $E_0$: | d | $E_1$: | d | $E_2$: | c |
| $E_3$: | c | $E_4$: | c | | |

Which states are the final (accepting) states? Select all that apply

(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 |
| [ˆ$r_1$] | 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**
$F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$
```