CMSC330 - Organization of Programming Languages
Spring 2025 - 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
- 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.	5
P3.	20
P4.	6
P5.	15
P6.	6
P7.	4
P8.	9
P9.	5
P10.	20
Total	100

Problem 1: Concepts

[Total 10 pts]

	True	False
`let f = fun x -> fun y -> [x y]` is an example of a higher order function	T	F
If you are at some state B in an FSM, the history of your path determines where you go next	T	F
If a function's type is `'a -> 'b -> int`, then the two inputs must be different type	T	F
In the expression `let x = 3 in let x = 4 in x`, only one variable binding occurs	T	F
`let f = print_string "hello"` will print the string "hello" everytime `f` is used	T	F
Regular Expressions can describe infinitely long strings	T	F
All compiled languages use explicit typing	T	F
An `accept` function that works for NFAs would also work for DFAs	T	F
OCaml is statically typed	T	F
In Ocaml, a multi-argument function is just a chain of single argument functions	T	F
In Ocaml, a multi-argument function is **not** equivalent to chaining single argument functions together	T	F

Problem 2: Project 2

[Total 5 pts]

Given a different implementation of the function `fold_tree`, which folds a tree into a list.

```
type tree = Node of tree * int * tree | Leaf

let rec fold_tree f b t =
  match t with
  | Leaf           -> b
  | Node (l, v, r) -> f (fold_tree f b r) [v] (fold_tree f b l)
```

Suppose that we write a **mystery** function that returns the traversal of the tree using tree fold.

```
let mystery t = fold_tree (fun x y z -> y @ z @ x) [] t
```

Describe the result of calling `mystery` on a tree? One sentence only.

It'd be pre-order.

Given a different implementation of the function `fold_tree`, which folds a tree into a list.

```
type tree = Node of tree * int * tree | Leaf

let rec fold_tree f b t =
  match t with
  | Leaf           -> b
  | Node (l, v, r) -> f [v] (fold_tree f b r) (fold_tree f b l)
```

Suppose that we write a **mystery** function that returns the traversal of the tree using tree fold.

```
let mystery t = fold_tree (fun x y z -> z @ x @ y) [] t
```

Describe the result of calling `mystery` on a tree? One sentence only.

It'd be in-order.

Here is a `fold_tree` implementation. It folds a tree into a list.

```
type tree = Node of tree * int * tree | Leaf

let rec fold_tree f b t =
  match t with
  | Leaf           -> b
  | Node (l, v, r) -> f [v] (fold_tree f b l) (fold_tree f b r)
```

Suppose we write a `post_order` function that returns the post_order traversal using tree fold.

```
let post_order tree = fold_tree (fun v l r -> l @ r @ v) [] tree
```

How would the result of calling `post_order` on a tree change if we change `fold_tree` to the following but leave everything else the same? One sentence only.

```
let rec fold_tree f b t =
  match t with
  | Leaf           -> b
  | Node (l, v, r) -> f (fold_tree f b r) [v] (fold_tree f b l)
```

It'd be pre-order.

Here is a `fold_tree` implementation. It folds a tree into a list.

```
type tree = Node of tree * int * tree | Leaf

let rec fold_tree f b t =
  match t with
  | Leaf           -> b
  | Node (l, v, r) -> f [v] (fold_tree f b l) (fold_tree f b r)
```

Suppose we write a `post_order` function that returns the post_order traversal using `fold_tree`.

```
let post_order tree = fold_tree (fun v l r -> l @ r @ v) [] tree
```

How would the result of calling `post_order` on a tree change if we change `fold_tree` to the following but leave everything else the same? One sentence only.

```
let rec fold_tree f b t =
  match t with
  | Leaf           -> b
  | Node (l, v, r) -> f (fold_tree f b l) (fold_tree f b r) [v]
```

It would be reverse in-order (right, root, left)

Problem 3: FSM and Regex

[Total 20 pts]

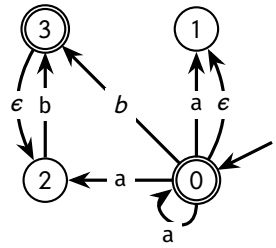
The following questions are independent from each other.

(a) Convert the following regular expression into an FSM (draw a box around your final answer):

`[ab]+c?` [15 pts]

(b) Acceptance [5 pts]

Given the following NFA, select all of the strings it accepts:



- (A) a
- (B) abbaabba
- (C) aaaa
- (D) abbba
- (E) abb
- (F) "" (empty string)

Problem 4: Regex

[Total 6 pts]

For an $\Sigma = \{a, b, c\}$, write a regular expression for strings that are in alphabetical order that have an even number of "a"s, an odd number of "b"s and any number of "c"s.

valid strings	**invalid strings**
`aab`	`abc`
`bbbc`	`aabb`
`b`	`abbc`
`aaaabbbccccccc`	`baac`

(aa)*b(bb)*c*

(xx)*y(yy)*z*

Problem 5: Debugging

[Total 15 pts]

We want to count the area codes from a list of phone numbers, . We use the following 3 functions to implement it.
1) The function **get_area_code** takes a phone number as an argument, returns **(Some area_code)** if the phone number is valid, otherwise returns **None**. A phone number's area code is the **first 3 digits**.

```
Valid Phone Number formats: (XXX)XXXXXXX or XXXXXXXXXX
 Examples:
    get_area_code "(1234567890" = None
    get_area_code "1234567890" = Some("123")
    get_area_code "hello" = None
    get_area_code = "(111)2223333" = Some("111")
```

```
(* string -> string Option *)
   let get_area_code phone_number =
1      let phone_re = "(?([0-9]{3}))?[0-9]{7}" in
       ... (* assume the rest works and uses the above regex to check the phone number *)
```

2) The function **update_count**, will update the counts of each area code in the database. If the area code does not exist, add it to the database. The database is represented as a `(string * int) list`.

```
Examples:
update_count [("122",1)] (Some "123") = [("122", 1); ("123", 1)]
update_count [] (Some "123") = [("123", 1)]
update_count [] None = []
update_count [("122",1)] None = [("122", 1)]
update_count [("122",1);("123",1)] (Some "123") = [("122", 1); ("123", 2)]
```

```
   let rec update_count db number =
       match db,number with
2           _,None -> db
3         |(num,count)::xs,Some(area) ->
4             if area = num then
5                 (num,count+1)::xs
6             else
7                 update_count xs number
8         |[],Some(area) -> [(area,0)]
```

3) The function **area_counts** a list of phone numbers and returns a `(string * int) list`. The `string` in the return type represents the area code of a phone number, and the `int` is the count of how many times that area code was in the list. If a phone number is not of valid format, the string is ignored.

```
Examples:
area_counts ["(123)4567890";"(098)7654321";"1234567890"] = [("123",2);("098",1)]
area_counts ["(111)2223333";"1114445555"] = [("111",2)]
area_counts [] = []
area_counts ["9998887777";"Malformed-ignored"] = [("999",1)]
```

```
    let area_counts lst =
9       fold_left (fun acc x ->
10          update_count acc (get_area_code x))
11      [] lst
```

There are at least **3 bugs present** in the lines with line numbers. Find them and fix them in the next page. Each bug should just require you to rewrite a single line of the program. If a line does not have a number next to it, then that line cannot be rewritten.

Note: Line is just to help grade, you will not get points just for identifying the line

(a) **Error 1** [5 pts]

Line: 1 Fix: (\([0-9]{3}\)|[0-9]{3})[0-9]{7}

(b) **Error 2** [5 pts]

Line: 7 Fix: (num,count)::(update_count xs number)

(c) **Error 3** [5 pts]

Line: 8 Fix: [(area,1)]

Problem 6: OCaml Typing [Total 6 pts]

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

```
let foo x y z = match x with
    a -> a
   |x::[] -> [z]
   |_ -> [7]
```

int list -> 'a -> int -> int list

```
let foo a b =
    map (a b) [1;2;3]
```

('a -> int -> 'b) -> 'a -> 'b list

Problem 7: Evaluation [Total 4 pts]

Evaluate the following OCaml expressions. If there is a compilation error, put "ERROR"

```
let foo f l =
    fold_left (fun a x -> ((f x)::a)) [] l in
foo (fun x -> x * 5) [1;2;3;4;5]
```

[25; 20; 15; 10; 5]

```
let foo =
    fun () -> let x = ref "hello" in
    fun a -> let res = !x in
        x := !x ^ a; res in
[foo () " World"; foo () " Everyone"]
```

["hello";"hello"]

Problem 8: Property Based Testing

[Total 9 pts]

Consider the following incorrect `filter` function for a list.

```
let rec filter f lst = match lst with
   [] -> []
  |x::xs -> if f x then (f x)::(filter f xs) else filter f xs
```

Consider the following property *p* about the `filter` function:

p : filtering a non-empty list with function *f* and filtering the same list with `not f` should result in 2 mutually exclusive lists

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

(●) Yes ( ) No

Using **our** implementation of `filter`, 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 lst = filter f lst <> filter (fun x -> not (f x)) lst
```

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

(●) Yes ( ) No

Consider the following property *p* about the `filter` function:

p : filtering a non-empty list with function *f* should always result in a shorter list than we started with

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

( ) Yes (●) No

Using **our** implementation of `filter`, 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 lst = List.length (filter f lst) < List.length (lst)
```

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 best possible fix. Each fix must only be used **once**, so choose the fix that fits the best.

Error	Answer
`Exception: Match_failure`	B
`Error: This expression has type 'a but an expression was expected of type 'a list`	D
`Error: This expression has type int but an expression was expected of type bool`	C
`Fatal Error: exception Failure("unimplemented")`	E
`Syntax Error`	A

	Fix
A	Make sure nested let expressions have a matching `in` keyword.
B	Check to make sure you are using exhaustive pattern matching
C	Make sure the guard of an if expression is the correct type
D	Make sure you are using cons and not @
E	Make sure you saved your code before testing

Match the following error messages with the best possible fix.

Error	Answer
`Error: This expression has type 'a but an expression was expected of type 'a list`	D
`Exception: Match_failure`	B
`Syntax Error`	A
`Error: This expression has type int but an expression was expected of type bool`	C
`Fatal Error: exception Failure("unimplemented")`	E

	Fix
A	Make sure nested let expressions have a matching `in` keyword.
B	Check to make sure you are using exhaustive pattern matching
C	Make sure the guard of an if expression is the correct type
D	Make sure you are using cons and not @
E	Make sure you saved your code before testing

Problem 10: Coding [Total 20 pts]

Restrictions for both 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. (Function #2 continues on the next page!)

(a) Query [15 pts]

Write a function called **huh** that takes in a `query` type and a value. Return **Some value** if the query is satisfied and **None** otherwise.

```
type 'a query = And of query * query | Not of query | Condition of ('a -> bool)

Examples:
The query q represents all values not > 4 and that are even
let q = And(Not(Condition(fun x -> x > 4)), Condition(fun x -> x mod 2 = 0))
huh q 4 = Some(4)
huh q 2 = Some(2)
huh q 8 = None

(* query -> 'a -> 'a Option *))
let rec huh query value =
    let rec work_huh query value = match query with
        |And(x,y) -> (work_huh x value) && (work_huh y value)
        |Not(x) -> not (work_huh x value)
        |Condition(f) -> f value in
    if work_huh query value then Some(value) else None
```

```
type 'a query = OR of query * query | Not of query | Condition of ('a -> bool)

Examples:
The query q represents all values not > 4 or that are even
let q = OR(Not(Condition(fun x -> x > 4)), Condition(fun x -> x mod 2 = 0))
huh q 4 = Some(4)
huh q 2 = Some(2)
huh q 7 = None

(* query -> 'a -> 'a Option *))
let rec huh query value =
    let rec work_huh query value = match query with
        |OR(x,y) -> (work_huh x value) || (work_huh y value)
        |Not(x) -> not (work_huh x value)
        |Condition(f) -> f value in
    if work_huh query value then Some(value) else None
```

(b) Tree [5 pts]

Suppose your above `huh` function works. Write a function called `query_tree` which takes in a query and a tree, then returns all the values that matches the query in a list.

```
type tree = Node of tree * int * tree | Leaf

Examples:
let q = And(Not(Condition(fun x -> x > 4)), Condition(fun x -> x mod 2 = 0))
let t = Node(Node(Leaf, 1, Leaf), -10, Node(Node(Leaf, 2, Leaf), 8, Leaf))
 -10
 / \
1   8
   /
   2
query_tree q t = [-10;2] (*order does not matter *)

Examples:
let q = OR(Not(Condition(fun x -> x > 4)), Condition(fun x -> x mod 2 = 0))
let t = Node(Node(Leaf, 1, Leaf), -10, Node(Node(Leaf, 2, Leaf), 7, Leaf))
 -10
 / \
1   7
   /
   2
query_tree q t = [-10;2;1] (*order does not matter *)

let rec query_tree q t = match t with
    let rec query_tree q t = match t with
    |Leaf -> []
    |Node(l,v,r) -> if huh q v = Some(v) then (query_tree q l) @ [v] @ (query_tree q r)
                                        else (query_tree q l) @ (query_tree q r);;
```

(c) get_nth_level [10 pts]

The first function will be `get_nth_level`. This function takes in a tree and a positive integer *n*. It will get all the values at the *n*th level of the tree and put them into a list.

```
(* Example tree: t
           1                <- level 1
          / \
     2           3          <- level 2
    / \         / \
  4     5     6     7       <- level 3
 / \   / \   / \   / \
8   9 10 11 12 13 14 15     <- level 4
 *)
get_nth_level t 1 = [1]
get_nth_level t 2 = [2;3]
get_nth_level t 3 = [4;5;6;7]
get_nth_level t 5 = []
(* Note: order of resulting list does not matter *)

type 'a tree = Petal | Stem of 'a tree * 'a * 'a tree

let rec get_nth_level tree n =
Solution
let rec get_nth_level tree n =
  let rec helper t n = match t with
  Petal -> []
  |Stem(l,v,r) -> if n = 0 then [v] else
                  helper l (n-1) @ helper r (n-1)
  in
  helper tree (n-1);;

-------------------------------------------------------------------------------------

type 'a tree = External | Internal of 'a tree * 'a * 'a tree

let rec get_nth_level tree n =
  let rec helper t n = match t with
  External -> []
  |Internal(l,v,r) -> if n = 0 then [v] else
                  helper l (n-1) @ helper r (n-1)
  in
  helper tree (n-1);;
```

(d) every_nth_level [10 pts]

The second function you will write is called `every_nth_level`. This function takes in a tree and a positive integer *n*. Return a list of all the values found at every *n*th level of the tree (please look at the examples below carefully). You may assume your `get_nth_level` function works and you may use it here.

```
(* using the same tree type *)
type 'a tree = Petal | Stem of 'a tree * 'a * 'a tree

(* using the same example tree
Example tree t
          1               <- level 1
         / \
     2           3        <- level 2
    / \         / \
  4     5     6     7      <- level 3
 / \   / \   / \   / \
8   9 10 11 12 13 14 15    <- level 4
*)
every_nth_level t 1 = [1;2;3;4;5;6;7;8;9;10;11;12;13;14;15];;
every_nth_level t 2 = [2;3;8;9;10;11;12;13;14;15];;
every_nth_level t 3 = [4;5;6;7];;
every_nth_level t 5 = [];;
(* Note: order of resulting list does not matter *)

'Using get_nth_level

let every_nth_level tree n =
  let rec loop i =
    if i mod n = 0 then
      match get_nth_level tree i with
      |[] -> []
      |x -> x @ (loop (i+1))
    else
      loop (i+1)
  in
  loop 1;;

---------
not using get_nth_level

let rec every_nth_level t n =
  let rec helper t curr = match t with
    |Petal -> []
    |Stem(l,v,r) -> let curr',add = if curr = 1 then n,[v] else (curr-1),[] in
                    add @ (helper l curr') @ (helper r curr')
  in
  helper t n

-------------------------------------------------------------------------------------------

let rec every_nth_level t n =
  let rec helper t curr = match t with
    |External -> []
    |Internal(l,v,r) -> let curr',add = if curr = 1 then n,[v] else (curr-1),[] in
                    add @ (helper l curr') @ (helper r curr')
```