CMSC330 Spring 2024 Quiz 2 Solutions

Problem 1: Basics

[Total 4 pts]

	True	False
Regular expressions can be used to parse text out of strings	(T)	F
When evaluating an expression, the order matters when there are side effects	(T)	F
The **concept** of fold is limited to lists	T	(F)
Map cannot be written in terms of fold_left	T	(F)
The **implementation** of fold_left is limited to lists	(T)	F
Regular Expressions cannot be used to parse text out of strings	T	(F)
Map can be written in terms of fold_right	(T)	F
The **concept** of fold is not limited to lists	(T)	F
Map cannot be written in terms of fold_right	T	(F)
Map can be written in terms of fold_left	(T)	F

Problem 2: Data and Map

[Total 8 pts]

Consider the following Variant from project 2:

```
type 'a tree = Leaf|BiNode of 'a tree * 'a * 'a tree (* left subtree, value, right subtree *)
```

Suppose we want to make a tree that looks like:

```
v1.     [1]        v2.      []         v3.      [6;7]         v4.     [9]
      /    \              /   \                /    \               /    \
   [4;5]   []          [0]   [2;3]          []     [8]          [8;7]    []
```

(a) How would you create a variable called `t` that is bound to a `int list tree` that corresponds to the above tree? [3 pts]

```
v1 - BiNode(BiNode(Leaf, [4; 5], Leaf), [1], BiNode(Leaf, [], Leaf))

v2 - BiNode(BiNode(Leaf, [0], Leaf), [], BiNode(Leaf, [2;3], Leaf))

v3 - BiNode(BiNode(Leaf, [], Leaf), [6; 7], BiNode(Leaf, [8], Leaf))

v4 - BiNode(BiNode(Leaf, [8;7], Leaf), [9], BiNode(Leaf, [], Leaf))
```

(b) Tree Map [5 pts]

Suppose we have a function called `tree map`. It works like `map`, but will map a `'a tree` to `'b tree`.

```
val tree_map f t: ('a -> 'b) -> 'a tree -> 'b tree
val map f l: ('a -> 'b) -> 'a list -> 'b list
```

v1. Using only `tree_map` and `map`, write a function that will add 5 to every element of the lists within a `int list tree`.

```
let addfive ltree =
    tree_map (fun a_list -> map (fun a -> a + 5) a_list) ltree
```

v2. Using only `tree_map` and `map`, write a function that will square every element of the lists within a `int list tree`.

```
let square ltree =
    tree_map (fun a_list -> map (fun a -> a * a) a_list) ltree
```

v3. Using only `tree_map` and `map`, write a function that will subtract 3 from every element of the lists within a `int list tree`.

```
let sub3 ltree =
    tree_map (fun a_list -> map (fun a -> a - 3) a_list) ltree
```

v4. Using only `tree_map` and `map`, write a function that will divide by 4 from every element of the lists within a `int list tree`.

```
let div4 ltree =
    tree_map (fun a_list -> map (fun a -> a / 4) a_list) ltree
```

Problem 3: Regex

[Total 8 pts]

Write a regex that describes a subset of valid umd emails. Emails take the form of a user's directory ID followed by the @ symbol, followed by one of the following domain names: `cs.umd.edu`, `terpmail.umd.edu`, or just `umd.edu`.

(a) Email Addresses [4 pts]

v1.

- A user's directory ID can be length 0 to length 8 consisting of only alphanumeric (both upper and lowercase) characters.
- A user's directory ID may not start with a digit.

```
^([a-zA-Z][a-zA-Z0-9]{0,7})?@((cs\.)|(terpmail\.))?umd\.edu$
```

v2.

- A user's directory ID can be length 2 to length 10 consisting of only alphanumeric (both upper and lowercase) characters.
- A user's directory ID may not start with a uppercase letter.

```
^[a-z0-9][a-zA-Z0-9]{1,9}@((cs\.)|(terpmail\.))?umd\.edu$
```

v3.

- A user's directory ID can be length 1 to length 6 consisting of only alphanumeric (both upper and lowercase) characters.
- A user's directory ID may not start with a lowercase letter.

```
^[A-Z0-9][a-zA-Z0-9]{0,5}@((cs\.)|(terpmail\.))?umd\.edu$
```

v4.

- A user's directory ID can be length 0 to length 10 consisting of only alphanumeric (both upper and lowercase) characters.
- A user's directory ID may not start with a an upper or lowercase letter.

```
^([0-9][a-zA-Z0-9]{0,9})?@((cs\.)|(terpmail\.))?umd\.edu$
```

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

v1.

```
^[Cliff|Anwar]+ is (great|the best)?$
```

(A) Cliff is sad (B) Anwar is the best (C) cliff is great

(D) winwrar is (E) flan is the best (F) the best

v2.

```
^[Cliff|Anwar]+ (is great|the best)?$
```

(A) Cliff is sad (B) Anwar is the best (C) cliff is great

(D) winwrar the best (E) flan (F) the best

v3.

```
^([Cliff|Anwar]+ is great|the best)?$
```

(A) Cliff is sad (B) Anwar is the best (C) cliff is great

(D) winwrar the best (E) flan is great (F) the best

v4.

```
^[Cliff|Anwar]+ (is|the) (great|best)?$
```

(A) Cliff is sad (B) Anwar is the best (C) cliff is great

(D) winwrar the best (E) flan is great (F) the best

(c) Which of the following regular expressions is not equivalent to the others? [2 pts]

v1.

(A) `[abc]+def?(g|hi)` (B) `(a|b|c)[abc]*def?g|(a|b|c)[abc]*def?hi`

(C) `(abc)+def(g|hi)|(abc)(abc)*de(g|hi)` (D) `[abc]+de((g|hi)|fg|fhi)`

(E) They are all the same

`(abc)+` is different from `[abc]+` or `(a|b|c)[abc]*`.

v2.

(A) `[abc]+def?(g|h)i` (B) `(a|b|c)[abc]*def?gi|(a|b|c)[abc]*def?hi`

(C) `[abc]+def(g|h)i|[abc]+de(g|h)i` (D) `[abc]+de((g|hi)|fg|fhi)`

(E) They are all the same

`(g|hi)|fg|fhi` doesn't account for `fgi` case.

v3.

(A) `[abc]+def?(g|hi)`

(B) `(a|b|c)[abc]*def?g|(a|b|c)[abc]*def?hi`

(C) `[abc]+def(g|hi)|[abc]+de(g|hi)`

(D) `[abc]+de((g|hi)|fg|fhi)`

(E) They are all the same

This might look similar to version 2, but we are looking for `g|hi` this time but v2 is looking for `(g|h)i`.

v4.

(A) `[abc]+def?(g|hi)`

(B) `(a|b|c)[abc]*def?g|(a|b|c)[abc]*dehi`

(C) `[abc]+def(g|hi)|[abc]+de(g|hi)`

(D) `[abc]+de((g|hi)|fg|fhi)`

(E) They are all the same

Doesn't account for `defhi` as the second `|` case is only `dehi`.