CMSC330 Spring 2023 Quiz 2
Quiz 2 Solutions

This entire quiz pertains to the OCaml Programming Language

Problem 1: Basics

[Total 3 pts]

Please circle **True** or **False** for the following statements:

OCaml uses a dynamic type system **True** **False**

While OCaml doesn't require you to declare types, types are associated with the variable and checked at compile time.

Variables can be overwritten in expressions **True** **False**

Variables cannot be overwritten in OCaml. Ex: If you say `let x = 2` and then say `let x = 3`, these are 2 separate instances of `x` and the value of the original `x` is not changed.

Lists in OCaml are allocated sequentially like Arrays in C **True** **False**

Lists in OCaml are dynamically allocated as linked lists. So there is not need to say at creation, how long the array should be.
In C, we had to do this: `int[5]`

Problem 2: Typing

[Total 8 pts]

For the following questions, provide the type of the given functions.

(a) `fun f a b -> ((f a) ^ "z") :: b` [2 pts]

- This is an anonymous function that has 3 parameters: `f`,`a`, and `b`
- **Type of f:** We can tell by the notation `(f a)` that `f` is a function that takes in the value `a`. Since the return value of `f` is used with the `^` operator and a string `"z"`, we can say it returns a string. The `^` operator can only be used between 2 strings.
- **Type of a:** There is no specific operators used with only `a` to indicate the type. Therefore we can assign it a generic type `'a`.
- **Type of b:** Since the `::` operator is only used between an element and a list, this must mean `b` is a list. We know that `(f a) ^ "z"` returns a string value; therefore `b` must be a `string list`.
- **Return Type:** Knowing that the `::` operator returns a list of the same type as elements in it, we can say the return type is a `string list` since `b` is a `string list`.
- Thus the type is `('a -> string) -> 'a -> string list -> string list`

(b)

`fun x -> fun y -> (x (y + 2)) +. 2.` [2 pts]

- This is an anonymous function that can be interpreted in 2 ways. This is a function that returns another function or a function that takes in 2 parameters.
- **Type of y:** Since y uses the + operator with the integer 2, this means that y is an integer. The + operator can only be used between 2 integers.
- **Type of x:** We can tell by the function application syntax that x is a function that takes in the value `(y + 2)` and returns a float. We know that `(y + 2)` is an int so we know x takes in an int. The reason we know the return type is a float is because after the return value the `+.` operator is used along with a float `2. .` The `+.` operator can only be used between 2 floats. Thus the type of x is `int -> float`.
- **Return Type:** We know that x`(y + 2)` and `2.` are floats which means a float `+.` a float returns a float.
- Thus the type is `(int -> float) -> int -> float`.

For the following questions, write an expression of the following types. All pattern matching must be exhaustive and you cannot use type annotations:

[2 pts] (c) `int -> float -> (int * float) list`

- This expression must be a function type. It takes in 2 arguments of type `int` and `float` and returns a `(int * float) list`
- **Return type:** We can tell the return type of the function should be `(int * float) list`.
- **Parameters:** We can assign variable names to all the other types and create our 2 parameters that way. The first parameter a has type int and the second parameter b has type float.
- **Showing a's type:** In order for the compiler to recognize that a is an integer, we should use an int operator on it, ex: `a + 1`. Operators like `+,-,/,*` are reserved for only between integers which is why the compiler recognizes a is an int.
- **Showing b's type:** Similarly, in order for the compiler to recognize that b is a float we should use float only operators such as `+., -., /., *.`. Something like `b +. 1.` would work.
- **Making the return type:** Working inside out, we should first try to construct an int and float tuple. Knowing `a + 1` is an int and `b +. 1.` is a float, combining them in a tuple `((a + 1, b +. 1.))` has the type `(int * float)`. To make this tuple into a list to match the return type we can add `[]` around them as such: `[(a + 1, b +. 1.)]`.
- Combining all these parts together we can create the function `fun a b -> [(a + 1, b +. 1.)]`.

[2 pts] (d) 'a -> 'b -> 'b -> 'a -> 'a list

- This expression must be a function type. It takes in 4 arguments of type `'a,'b,'b,'a`, and returns a `'a list`,
- **Return Type:** The last type is always the return type; in this case `'a list`.
- **Parameters:** We can assign variable names to all the other types and create the 4 parameters. The first parameter a has type 'a, the second parameter b has the type 'b, the third parameter c has type 'b, and the fourth parameter d has type 'a.
- **Showing a and d's type:** In order to show that a and d are the same type we can use any comparative operator to show they are the same type, ex: `a = d, a > d, a < d`. You cannot use operators such as `+, -, /`, etc to show they are the same type because these are operators reserved for only int types.
- **Showing b and c's type:** Similarly in order to show that b and c are the same type we can use another comparative operator, ex: `b = c`.
- **Making the return type:** Since comparative operators return boolean values, the most likely expression you want to use is an if statement. Something like `if a = d || b = c` since it combines both comparisons. Finally in order to return `'a list` we should use the 2 parameters that have an `'a` type to return an `'a list`, ex: `[a], [d], [d;a]`, etc.
- Combining these parts we can create something like: `fun a b c d -> if a = d || b = c then [a] else [d].`

For questions 3 and 4, you may use the following higher order functions:

```
let rec fold f a l = match l with [] -> a | h::t -> fold f (f a h) t
let rec foldr f l a = match l with[] -> a | h::t -> f h (foldr f t a)
let rec map f l = match l with[] -> [] | h::t -> (f h)::(map f t)
```

[Total 4 pts]

Problem 3: Code Completion

Write a function called `parity_sum` that takes in an `int list` and returns an `int * int * int tuple` where the first value in the tuple is the sum of even indices, the second is the sum of the odd indices and the third value is the size of the list. Lists are 0 indexed.

Example:

```
parity_sum [1;2;3;4] = (4,6,4)
parity_sum [1;10;2;20;3;30] = (6,60,6)
parity_sum [] = (0,0,0)
parity_sum [-1;-5;1,5] = (0,0,4)
```

```
let partity_sum lst = fold (fun acc x -> ___Blank_1___ ) ___Blank 2___ arr
```

Blank 1 (3 pts):

- **Length of list:** First, let's focus on finding the length of the list. If we iterate through each of the elements, we can add 1 to the total count of elements which at the end would return the length of the list. We need to do this to only the third element of the accumulator.
- **Adding odd/even index only:** The third element of the accumulator tells us the length of the list up to the current element, which is basically the index of the list; thus, we can use that to find out if we are at an even or odd index.
- **Combining:** We can match the `acc` to the tuple `(a, b, c)` to extract each individual element. Then we can check if `c mod 2 = 0` to see if we are at an even or odd index and add `x` to `a` or `b` appropriately.
- This results in `match acc with (a, b, c) -> if c mod 2 = 0 then (a + x, b, c + 1) else (a, b + x, c + 1)`.

Blank 2 (2pts) :

Since this is the accumulator value that is given to the anonymous function this is also the base case return value. In this case the base case would be `arr = []` which means the function would return `(0, 0, 0)`; therefore this blank should be `(0, 0, 0)`.

Additional Notes:

- In problems like this it is usually easier to start with the value of the accumulator. Since the starting accumulator is the return value for the base case (usually an empty list) this is the value you want to return if you called the fold on an empty list.
- It's easier to come up with the body of the anonymous function if you think about what to specifically do for only one element of the list. This element is generalized to `x` in the anonymous function.
- The return value of the anonymous function is the new accumulator that is used for the next element in the list. This is also the return value of the fold call as a whole so think about what specifically you want the fold call to return.

Problem 4: Coding

[Total 5 pts]

`Duplicate`

Create a function `dup_elems lst n` that duplicates every element in `lst` n times

Example:

```
dup_elems [1;2;3;4] 1 = [1;1;2;2;3;3;4;4]
dup_elems [] 2 = []
dup_elems ["a";"a";"b";"c"] 0 = ["a";"a";"b";"c"]
```

- For question 4, there are a couple of approaches we can take (both using recursion exclusively, or using both higher order functions (fold) and recursion).
- Let's start by analyzing the different examples/cases we're given. When we pass in an empty list, we return an empty list, regardless of what n is. If `n = 0`, there are no duplicates, and we return the original list. For `n >= 1`, we repeat each element in our `lst n` times. In all cases, the result is a list.
- **Helper Function:**
  - **Purpose:** If `n >= 1`, we need to be able to have something in our code that allows us to be able to repeat the number of elements. This can be done more clearly through a helper function, which takes in the element we want to repeat, and how many times we want to repeat it.

- **Return Type:** Now, the harder part: what do we want our helper function to return? Since we want to return multiple of the same elements, the best return value would be the repeated elements in a list. We wouldn't want to return a single integer, since that doesn't make sense for multiple elements, or a tuple, since we don't know how many elements we want to repeat (we can't make a fixed size tuple if we're calling the helper with multiple values for n).
- **Solution:** Since there's no restriction on using the @ operator in this question, this can help us in writing our solution. In both cases, we want our helper function to duplicate a given element n times.

```
let rec dup_helper elem n =
match n with
| 0 -> [elem]
| _ -> elem ::  dup_helper elem (n - 1)
```

- In this helper, if we hit the base case where `n = 0`, we return the provided element, as we don't want it to be repeated. In all other cases of n (we use the wildcard here), we add the element to our growing list, and recursively call our helper.

Now that we have our helper, we can take a recursive approach, or use fold.

- **With recursion:**
  - ```
    let rec duplicate lst n = match lst with
    | [] -> []
    | h::t -> (dup_helper h n) @ duplicate t n
    ```
  - Here, we only need to do an explicit check on the list whether to return an empty list. We don't need to check the value of n, since that is handled in the recursive helper
- **With fold:**
  - `let duplicate lst n = fold (fun a x -> a @ dup_helper x n) [] lst`
  - Similar to the recursive approach, we look at every element in the list, and call the recursive helper with each element and n, and append that to our accumulator (which is a list).
    ```