CSC 330
Programming Languages
Assignment 3

Note 1 **This assignment is to be done individually**

Note 2 You can discuss the assignment with others, but sharing and copying code is prohibited.

Errata:

- 2021/02/08: Corrected the name of the functions: `all_answers`, `first_answer` in their descriptions.

A note on Academic Integrity and Plagiarim

Please review the following documents:

- Standards for Professional Behaviour, Faculty of Engineering:
  `https://www.uvic.ca/engineering/assets/docs/professional-behaviour.pdf`
- Policies Academic Integrity, UVic:
  `https://www.uvic.ca/students/academics/academic-integrity/`
- Uvic's Calendar section on Plagirism:
  `https://www.uvic.ca/calendar/undergrad/index.php#/policy/Sk_0xsM_V`

Note specifically:

> Plagiarism
> Single or multiple instances of inadequate attribution of sources should result in a failing grade for the work. A largely or fully plagiarized piece of work should result in a grade of F for the course.

You are responsible for your own submission, but you could also be responsible if somebody plagiarizes your submission.

Objectives

After completing this assignment, you will have experience:

- Learn to use higher-order functions
- Learn to use `map`, `filter`, `foldl`
- Learn to use currying

Your task, should you choose to accept it

Part 1

This time we are going to build a tree data structure. The datatype declaration is:

```
datatype tree = emptyTree |
                nodeTree of int * tree * tree
```

In this case the tree is going to be a binary tree of integers. The tree should be kept in order. Specifically, the nodes to the left should less or equal than the current node, and the ones to the right bigger than the current node. Yes, the tree can have duplicates.

Implement the following functions. Note that some functions are curried.

1. `tree_insert_in_order(t, v)`. Insert in order. If the node is duplicated, insert to the left. This is a recursive function.

2. `tree_delete(t,v)`. First, find the node to delete (if v appears more than once, find the first instance). If v does not exist, raise `NotFound`. Second, if the node to delete has one child only, then simply delete the node and reconnect the child to the parent; otherwise, find the maximum node in the left child, remove it from this subtree, and create a note to replace the one you are deleting (with the new subtree as its left child). Use `tree_max` and a recursive call to `tree_delete`.

3. `tree_height t`. Return the maximum height of the tree. 0 for `EmptyTree`. Use recursion.

4. `tree_fold_pre_order f acc t`. Write a "fold" function that traverses the tree in preorder (node, left children, right children) "folding" the tree using the function `f`. Use `acc` as the starting value.

5. `tree_max t`. Find the maximum value in the tree (returns an option). Use a `val` expression and `tree_fold_pre_order` to write this function.I know, I know, this is the most inefficient way to find the maximum, and I agree; we are learning functional programming, not algorithms. **Restriction: define it using val and use `tree_fold_pre_order`**

6. `tree_to_list t`. Convert the tree to a list, in pre-order. **Restriction: define it using val and use `tree_fold_pre_order`**

7. `tree_filter f t`. Write a function to "filter" the tree. The function should return a tree with only nodes for which `f` returns true. Use `tree_delete`. It will result in a very simple implementation (though inefficient).

8. `tree_sum_even t`. Using a `val` expression, `tree_filter` and `tree_fold_pre_order`, and function composition to write a function that sums the nodes that are are even. Use the `mod` operator. **Restriction: use function composition and a val expression to define this function**.

Part 2

Write the following two functions. You will need them in subsequent functions.

1. Write a function `first_answer f lst` that has type `('a -> 'b option) -> 'a list -> 'b` (notice that the 2 arguments are curried. `f` should be applied to elements `lst` in order, until the first time `f` returns `SOME v` for some v; in that case v is the result of the function. If `f` returns `NONE` for all list elements, then `first_answer` should raise the exception `NoAnswer`.

2. Write a function `all_answer f lst` of type `('a -> 'b list option) -> 'a list -> 'b list option` (notice the 2 arguments are curried). Like `first_answer`. `f` should be applied to elements of the second argument. If it returns `NONE` for any element, then `all_answer` returns `NONE`. Otherwise the calls to `f` will have produced `SOME lst1, SOME lst2, ... SOME lstn` and the result of `all_answer` is `SOME lst` where `lst` is `lst1, lst2, ..., lstn` (the order in the result list should be preserved). Note that `all_answer f []` should return `SOME []` for any `f`.

For the rest of this part, the main the goal is to implement pattern matching (very similar to how SML does it). We will use the following datatypes:

```
datatype pattern = Wildcard | Variable of string | UnitP | ConstP of int
    | TupleP of pattern list | ConstructorP of string * pattern

datatype value = Const of int | Unit | Tuple of value list
    | Constructor of string * value
```

Given `value v` and `pattern p`, either `p` matches `v` or not. If it does, the match produces a list of `string * value` pairs; order in the list should be preserved. This is similar to the following SML expression:

```
case v of
   p (a, b, c, ... )=>
```

The value v is the one to match P is the pattern that might create zero or more some bindings `a, b`, etc. Your function will try to match the value `v` to the pattern `p`. If it does not match, it will return NONE. If it matches it will return the list of bindings `("a", val1), ("b", val2), ("c", val3), ....` The number of bindings will depend on the pattern.

The rules for matching are (they are the same as in SML):

- `Wildcard` matches everything and produces the empty list of bindings.
- `Variable s` matches any value v and produces the one-element list holding `(s,v)`.
- `UnitP` matches only `Unit` and produces the empty list of bindings.
- `ConstP i` matches only `Const i` (for any integer i) and produces the empty list of bindings.
- `TupleP ps` matches a value of the form `Tuple vs` if `ps` and `vs` have the same length and for all i, the i-th element of `ps` matches the i-th element of `vs`. The list of bindings produced is all the lists from the nested pattern matches appended together.
- `ConstructorP(s1,p)` matches `Constructor(s2,v)` if `s1` and `s2` are the same string (you can compare them with =) and `p` matches `v`. The list of bindings produced is the list from the nested pattern match. We call the strings `s1` and `s2` the *constructor name*.

- Nothing else matches.

3. Write a function `check_pattern` that takes a pattern and returns true if and only if all the variables appearing in the pattern are distinct from each other (i.e., use different strings). The constructor names are not relevant. Hints: use two helper functions. The first takes a pattern and returns a list of all the strings it uses for variables (use `foldl` with a function that uses `append`). The second helper function a list of strings and decides if it has repeats (`List.exists` may be useful).

4. Write a function `match` that takes a `value * pattern` and returns a `(string * value) list option`, namely `NONE` if the pattern does not match and `SOME lst` where `lst` is the list of bindings if it does. Note that if the value matches but the pattern has no patterns of the form `Variable s`, then the result is `SOME []`. **Remember to look above for the rules for what patterns match what values, and what bindings they produce**. Hints: use a case expression with 7 branches (one per rule). The branch for tuples uses `all_answer` and `ListPair.zip`.

5. Write a function `first_match` that takes a value and a list of patterns and returns a `(string * value) list option`, namely `NONE` if no pattern in the list matches or `SOME lst` where `lst` is the list of bindings for the first pattern in the list that matches. Use `first_answer` and a handle-expression. Notice that the 2 arguments are curried.

Hints

These are the bindings your program should generate (not necessarily in the same order as described above):

1.
```
val tree_insert_in_order : tree * int -> tree
val tree_delete : tree * int -> tree
val tree_height : tree -> int
val tree_fold_pre_order : (int * 'a -> 'a) -> 'a -> tree -> 'a
val tree_max : tree -> int option
val tree_to_list : tree -> int list
val tree_filter : (int -> bool) -> tree -> tree
val tree_sum_even : tree -> int
```

2.
```
val first_answer : ('a -> 'b option) -> 'a list -> 'b
val all_answers : ('a -> 'b list option) -> 'a list -> 'b list option
val check_pattern : pattern -> bool
val match : value * pattern -> (string * value) list option
val first_match : value -> pattern list -> (string * value) list option
```

Tests provided

The file `main.sml` contains some tests that will clarify the behaviour of your functions. I recommend you modify this file to add your own tests. **You will not submit this file**. Make sure you test your submission with an unmodified version of this file to make sure your program works as expected. You can exchange tests with other students (but you are forbidden from sharing the solutions to the assignment).

Restrictions

The only restrictions to this assignment are described in the description of the functions.

Evaluation

Solutions should:

1. Compile and run in `linux.csc.uvic.ca` If a program does not compile **for any reason** it will receive a zero.
2. Compile with no warnings. A warning will receive an automatic zero.
3. Satisfy the restrictions above. You will lose marks if you don not follow them.
4. Pass tests. Your grade will be determined based on how many tests it passes.

Second submission

If you choose, you can submit this assignment a second time (the deadline will be set after the results of the first submission are published). The final grade of your assignment will be:

$$max(first\_submission, 0.5 * second\_submission)$$

You are not required to submit a second time.

What to submit

1. Using conneX, submit your version of the file `patterns.sml`. **Do not submit any other files**.
2. You should assume that your submission was received correctly only if both of the following two conditions hold:
   (a) You receive a confirmation email from conneX.
   (b) You can download and view your submission, and it contains the correct data.
3. In the event that your submission is not received by the marker, you will need to demonstrate that both of the above conditions were met if you want to argue that there was a submission error.