CPSC 326: Homework Assignment 9 *Due: Thursday, May 4*

The goal of this assignment is to gain practice writing more OCaml functions. Note that you are free to use whatever IDE and machine you prefer for this assignment. However, to complete the assignment, you will need to download and install OCaml and/or use the the remote development server (`ada.gonzaga.edu`), which has OCaml installed already.

Overview:

1. Use the GitHub Classroom link (posted in Piazza) to copy the starter code into your own repository. Clone the repository in the directory where you will be working on the assignment (e.g., onto `ada` or your own machine).
2. Write the functions described below in `hw9.ml`.
3. Write multiple tests for each function to ensure correctness within the `hw9_tests.ml` file.
4. Submit your two program files. Be sure to add, commit, and push all assignment files to your GitHub repo. You can verify that your work has been submitted via the GitHub page for your repo.

Instructions: Implement the following functions *from scratch* in `hw9.ml` (i.e., without just calling functions provided by OCaml). In addition you must:

- only use the OCaml constructs we've discussed so far in class or as provided in the function description (if you go beyond what we've done, you'll receive no points for the question);
- *not use any* `if-then-else` constructs for conditionals (i.e., you *must* use pattern matching and guards as necessary);
- follow the general style guide provided by OCaml (https://ocaml.org/learn/tutorials/guidelines.html)
- appropriately comment your code throughout including a file header with your name, file name, the date, and a brief description; and
- create sufficient test cases for your functions to ensure they work correctly.

The following functions must be implemented as stated above. If you have questions on how any of the following are supposed to work, please ask either during class or on piazza:

1. Rewrite your function `my_last` from HW-8 so that it uses pattern matching instead of if-then-else.
2. Rewrite your function `my_init` from HW-8 so that it uses pattern matching instead of if-then-else.
3. Rewrite your function `my_replace` from HW-8 so that it uses pattern matching with guards instead of if-then-else.
4. Rewrite your function `my_replace_all` from HW-8 so that it uses pattern matching instead of if-then-else.

5. Rewrite your function `my_elem_sum` from HW-8 so that it uses pattern matching with guards instead of if-then-else.

6. Rewrite your function `my_range_sum v1 v2` from HW-8 so that it uses pattern matching with guards instead of if-then-else.

For the remaining questions, the goal is to implement basic functions for a "key-value pair" collection algebraic data type. Thus, you will need to add the following to your `hw9.ml` *file. Finally, note that* α*,* β*,* γ *and* δ *are used for* `'a`*,* `'b`*,* `'c`*, and* `'d` *in the problems below.*

```
type ('a, 'b) kvlist = Node of 'a * 'b * ('a, 'b) kvlist
                     | Nil ;;
```

7. Write a function `insert` with type $\alpha \to \beta \to (\alpha, \beta)$ `kvlist` $\to (\alpha, \beta)$ `kvlist`. Examples:
   - `insert 'a' 1 Nil` ⇒ `Node ('a', 1, Nil)`
   - `insert 'a' 1 (Node ('b', 1, Nil))` ⇒ `Node ('a', 1, Node ('b', 1, Nil))`
8. Write a function `remove` with type $\alpha \to (\alpha, \beta)$ `kvlist` $\to (\alpha, \beta)$ `kvlist` that removes all key-value pairs in a collection that have a given key. Examples:
   - `remove 'a' Nil` ⇒ `Nil`
   - `remove 'a' (Node ('a', 1, Nil))` ⇒ `Nil`
   - `remove 'a' (Node ('a', 1, Node ('a', 2, Nil)))` ⇒ `Nil`
   - `remove 'a' (Node ('b', 1, Node ('a', 2, Nil)))` ⇒ `Node ('b', 1, Nil)`
9. Write a function `size` with type (α, β) `kvlist` $\to$ `int`. The size function should return the number of key-value pairs in the collection, where `size Nil` is 0.
10. Write a function `has_key` with type $\alpha \to (\alpha, \beta)$ `kvlist` $\to$ `bool`, which returns true if the collection contains a key-value pair with the given key, and false otherwise.
11. Write a function `key_values` with type $\alpha \to (\alpha, \beta)$ `kvlist` $\to \beta$ `list`. This function should return a list of the values for a given key in a collection. Examples:
    - `key_values 'a' Nil` ⇒ `[]`
    - `key_values 'a' (Node ('a', 1, Node ('b', 2, Nil)))` ⇒ `[1]`
    - `key_values 'a' (Node ('a', 2, Node ('a', 3, Nil)))` ⇒ `[2; 3]`
    - `key_values 'c' (Node ('a', 1, Node ('b', 2, Nil)))` ⇒ `[]`
12. Write a function `combine` with type (α, β) `kvlist` $\to (\alpha, \beta)$ `kvlist` $\to (\alpha, \beta)$ `kvlist`. This function should work the same as `(@)` but for key-value collections.
13. Write a function `group` with type (α, β) `kvlist` $\to (\alpha, \beta$ `list`$)$ `kvlist`. This function should combine key-value pairs with duplicate keys. Examples:

- `group Nil` ⇒ `Nil`
- `group (Node ('a', 1, Nil))` ⇒ `Node ('a', [1], Nil)`
- `group (Node ('a', 1, Node ('b', 2, Nil)))` ⇒ `Node ('a', [1], Node ('b', [2], Nil))`
- `group (Node ('a', 1, Node ('a', 2, Nil)))` ⇒ `Node ('a', [1; 2], Nil)`

14. Write a function `invert` with type (α, β) `kvlist` $\rightarrow (\beta, \alpha)$ `kvlist`. This function simply "flips" each key-value pair in the collection. Examples:

    - `invert Nil` ⇒ `Nil`
    - `invert (Node ('a', 1, Nil))` ⇒ `Node (1, 'a', Nil)`
    - `invert (Node ('a', 1, Node ('b', 2, Nil)))` ⇒ `Node (1, 'a', Node (2, 'b', Nil))`

15. Write a function `kv_filter` with type $(\alpha \rightarrow \beta \rightarrow$ `bool`$) \rightarrow (\alpha, \beta)$ `kvlist` $\rightarrow (\alpha, \beta)$ `kvlist`. This function should be identical to the `filter` function but work over `kvlist` values as opposed to lists.

16. Write a function `kv_map` with type $(\alpha \rightarrow \beta \rightarrow \gamma * \delta) \rightarrow (\alpha, \beta)$ `kvlist` $\rightarrow (\gamma, \delta)$ `kvlist`. This function should be identical to the `map` function but work over `kvlist` values as opposed to lists.

17. Write a function `count_keys_by_val` with type `int` $\rightarrow (\alpha, \beta)$ `kvlist` $\rightarrow (\beta,$ `int`$)$ `kvlist`. The first parameter is a "threshold" value. The function returns the number of key-value pairs each value is associated such that the number of key-value pairs is larger than the threshold value. Your function should be a "one-liner" constructed from the functions defined above, including the use `kv_map` and `kv_filter`. You can also use the `List.length` function in your implementation. Note that you can also use a `let-in` expression (to break the one-liner into parts). Examples:

    - `count_keys_by_val 0 Nil` ⇒ `Nil`
    - `count_keys_by_val 0 (Node ('a', 1, Nil))` ⇒ `Node (1, 1, Nil)`
    - `count_keys_by_val 1 (Node ('a', 1, Nil))` ⇒ `Nil`
    - `count_keys_by_val 1 (Node ('a', 1, Node ('b', 1, Nil)))` ⇒ `Node (1, 2, Nil)`
    - `count_keys_by_val 1 (Node ('a', 1, Node ('b', 2, Nil)))` ⇒ `Nil`

Testing. Like in HW-8 we will use our own basic testing framework to test the HW-9 functions above. Your `hw9_test.ml` program should be structured as follows.

```
(*
   Name: <your-name-here>
   File: hw9_tests.ml
   Date: Spring 2023
   Desc: HW9 function unit tests
*)

open Hw9

let msg = "--- Running HW8 Tests --- ";;
```

```
print_endline msg;;

(* For equality assertions *)
let assert_equal v1 v2 msg =
  let cond = v1 = v2 in
  assert (if not cond then print_endline ("TEST FAILED: " ^ msg) ; cond)
;;

(* Question 1: my_last tests *)
...

(* Question 2: my_init tests *)
...

...
```

To check that your program "passes" the tests, from the command line you will first need to compile your program:

```
ocamlopt -o hw9_tests hw9.ml hw9_tests.ml
```

You can then run the executable:

```
./hw9_tests
```

If your tests succeed you should not get any additional output (other than the "Running HW9 Tests" message.

Homework Submission and Grading. Your homework will be graded using the files you have pushed to your GitHub repository. Thus, you must ensure that all of the files needed to compile and run your code have been successfully pushed to your GitHub repo for the assignment. This homework assignment is worth a total of **20 points**. In particular, you will receive one point per function and an additional three points for your unit tests. Note that you do not need to create a writeup for this assignment.