CPSC 326: Homework 9 *Due: Tuesday, Apr. 20*

Reading Assignment. Read the following sections in the textbook:

- Sections 12.1–12.3: LP Deduction, Syntax, and Unification

Programming Homework. The goal of this assignment is to implement a basic key-value pair collection algebraic data type:

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

Your job is to implement and test each of the following functions. You must write each function "from scratch" (unless otherwise specified) and using using only what we have discussed in class. Each function should use recursion as appropriate and you cannot use "*if-then-else*" in your function implementations. Finally, you must:

- 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, including error cases as needed.

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. Note that α, β, γ and δ are used for `'a`, `'b`, `'c`, and `'d` below.

1. Write a function `insert` with type $\alpha \to \beta \to (\alpha, \beta)$ `kvlist` $\to (\alpha, \beta)$ `kvlist`. Examples:
   - `insert 'a' 1 Nil` $\Rightarrow$ `Node ('a', 1, Nil)`
   - `insert 'a' 1 (Node ('b', 1, Nil))` $\Rightarrow$ `Node ('a', 1, Node ('b', 1, Nil))`
2. 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` $\Rightarrow$ `Nil`
   - `remove 'a' (Node ('a', 1, Nil))` $\Rightarrow$ `Nil`
   - `remove 'a' (Node ('a', 1, Node ('a', 2, Nil)))` $\Rightarrow$ `Nil`
   - `remove 'a' (Node ('b', 1, Node ('a', 2, Nil)))` $\Rightarrow$ `Node ('b', 1, Nil)`
3. 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.

4. Write a function `has_key` with type $\alpha \rightarrow (\alpha, \beta)$ `kvlist` $\rightarrow$ `bool`, which returns true if the collection contains a key-value pair with the given key, and false otherwise.

5. Write a function `keys` with type (α, β) `kvlist` $\rightarrow \alpha$ `list`. This function should return a list of the keys in a given collection. Examples:

   - `keys Nil` $\Rightarrow$ `[]`
   - `keys (Node ('a', 1, Node ('b', 2, Nil)))` $\Rightarrow$ `['a'; 'b']`
   - `keys (Node ('a', 1, Node ('a', 2, Nil)))` $\Rightarrow$ `['a'; 'a']`

6. Write a function `values` with type (α, β) `kvlist` $\rightarrow \beta$ `list`. This function should return a list of the values in a given collection. Examples:

   - `keys Nil` $\Rightarrow$ `[]`
   - `keys (Node ('a', 1, Node ('b', 2, Nil)))` $\Rightarrow$ `[1; 2]`
   - `keys (Node ('a', 2, Node ('a', 2, Nil)))` $\Rightarrow$ `[2; 2]`

7. Write a function `key_values` with type $\alpha \rightarrow (\alpha, \beta)$ `kvlist` $\rightarrow \beta$ `list`. This function should return a list of the values for a given key in a collection. Examples:

   - `key_values 'a' Nil` $\Rightarrow$ `[]`
   - `key_values 'a' (Node ('a', 1, Node ('b', 2, Nil)))` $\Rightarrow$ `[1]`
   - `key_values 'a' (Node ('a', 2, Node ('a', 3, Nil)))` $\Rightarrow$ `[2; 3]`
   - `key_values 'c' (Node ('a', 1, Node ('b', 2, Nil)))` $\Rightarrow$ `[]`

8. Write a function `combine` with type (α, β) `kvlist` $\rightarrow (\alpha, \beta)$ `kvlist` $\rightarrow (\alpha, \beta)$ `kvlist`. This function should work the same as `(@)` but for key-value collections.

9. 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` $\Rightarrow$ `Nil`
   - `invert (Node ('a', 1, Nil))` $\Rightarrow$ `Node (1, 'a', Nil)`
   - `invert (Node ('a',1, Node ('b',2,Nil)))` $\Rightarrow$ `Node (1,'a', Node (2,'b',Nil))`

10. Write a function `group` with type (α, β) `kvlist` $\rightarrow (\alpha, \beta$ `list`$)$ `kvlist`. This function should combine key-value pairs with duplicate keys. Examples:

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

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

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

13. Write a function `join` with type (α, β) `kvlist` $\to (\alpha, \beta)$ `kvlist` $\to (\alpha, \beta$ `list`$)$ `kvlist`. This function should do the following: (1) group the two kvlists; (2) combine the grouped lists from 1; (3) group the result from 2; and then (4) unnest each element of the result of 3. You must only call the functions you've already written above, but will need to write an `unnest` helper function using a `let-in` expression. (In other words, this is essentially a one-liner with your helper function and functions you've already written above.) Your `unnest` helper function can use the `List.flatten` function. Examples:

    - `join Nil Nil` $\Rightarrow$ `Nil`
    - `join (Node ('a', 1, Nil)) (Node ('a', 2, Nil))` $\Rightarrow$ `Node ('a', [1;2], Nil)`
    - `group (Node ('a',1,Nil)) (Node ('b',2,Nil))` $\Rightarrow$ `Node ('a',[1],Node ('b',[2],Nil))`

14. Write a function `count_keys_by_val` with type `int` $\to (\alpha, \beta)$ `kvlist` $\to (\beta,$ `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 equal to or larger than the threshold. 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 1 Nil` $\Rightarrow$ `Nil`
    - `count_keys_by_val 1 (Node ('a', 1, Nil))` $\Rightarrow$ `Node (1, 1, Nil)`
    - `count_keys_by_val 2 (Node ('a', 1, Nil))` $\Rightarrow$ `Nil`
    - `count_keys_by_val 1 (Node ('a', 1, Node ('b', 1, Nil)))` $\Rightarrow$ `Node (1, 2, Nil)`
    - `count_keys_by_val 1 (Node ('a', 1, Node ('b', 2, Nil)))` $\Rightarrow$ `Node (1, 1, Node (2, 1, Nil))`

Finally, you will need to test each of your functions above using the same approach as in HW-7 and HW-8.

Homework Submission. All homework must be submitted through GitHub Classroom. A link for each assignment will be posted on Piazza when the homework is assigned. Be sure all of your code is pushed by the due date (which you can double check for your repository using the GitHub

website). Each programming assignment is worth **35 points**. The points are allocated based on the following.

- **Correct and Complete (25 points)**. Your code must correctly and completely do the requested tasks using the requested techniques. Note that for most assignments you will be provided a *partial* set of test cases to help you determine a *minimal level* of correctness. If your program fails any of the provided test cases you will only receive partial credit. *Note that passing the given test cases does not mean your work is complete nor correct.* Your assignment will also be graded with additional test cases (not provided to you) that will help the graders determine the extent of your solution and your final score. Note that for C++ code, correctness also implies properly handling the creation and deletion of dynamic memory (i.e., the absence of memory leaks).

- **Evidence and Quality of Testing (5 points)**. As part of your homework assignments you must develop additional test cases beyond those given to you to ensure your program is correct and complete. These test cases must be turned in with your assignment. You will be graded on the scope and quality of the additional test cases you provide.

- **Formatting and Comments (5 points)**. Your code must be formatted consistently and appropriately for the language used. For C++, you must follow the provided style guide (see the course webpage).You must also comment your code and test cases, which at a minimum must include a file heading (see examples provided), function comments, and meaningfully selected variable, class, and function names. See the assignment for style guides in other languages.