Assignment 3
Due: June 22, 2023 at 11:59PM

1 Assignment 3 Specification

1.1 Overview

In this assignment, you will implement more database operations, starting from your lab3 work. In addition, you will increase efficiency of some algorithms by building tail-recursive versions of them.

A list reversal function using the `snoc` list function can be written as follows:

```
let rec snoc a L =
  match L with
    | [] -> [a]
    | h::t -> h::snoc a t;;

let rec rev L =
  match L with
    | [] -> []
    | h::t -> snoc h (rev t);;
```

The complexity of `rev` is $O(n^2)$ for a list of length n because `rev` invokes `snoc` and `snoc` recurses on each List element for a total of:

$n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$ operations.

For example: `snoc 4 [1;2;3]` generates the following recursive calls:

```
snoc 4 [1;2;3];;      // [1;2;3;4]
snoc 4 [2;3];;        // [2;3;4]
snoc 4 [3];;          // [3;4]
snoc 4 [];;           // [4]
```

Then to reverse a list of 3 elements `snoc` is invoked on lists of length $3+2+1=6$ times, on length 4 lists $4+3+2+1=10$ times, etc. The number of operations is $\frac{n^2+n}{2}$, giving `rev` a compexity of $O(n^2)$.

We can do better. A tail recursive reverse of $O(n)$ for a list of length n is:

```
let rec helper a b =
  match a with
    | [] -> b
    | h::t -> helper t (h::b);;

let rev L = helper L [];;

rev [1;2;3;4];; // returns list = [4; 3; 2; 1]
```

The real work is done through the helper function by using tail recursion to accumulate and eventually returning the result of `h::a`. To reverse a list of length n now requires only n recursive calls on the helper function. Some examples of the intermediate calls to helper:

```
helper [] [3;2;1];; // [3;2;1] the base case
helper [1] [];;     // returns (helper [] 1::[]) or helper [] [1]
helper [2;3] [1];;  // returns helper [3] 2::[1] or helper [3] [2;1]
                    //                            = helper [] 3::[2;1]
                    //                            = helper [] [3;2;1]
                    //                            = [3;2;1]
```

To reverse 1000 elements by the $O(n)$ `rev` requires $\sim$ 1000 calls.

To reverse 1000 elements by the $O(n^2)$ `rev` requires $\sim$ 1,000,000 calls.

1.2 Implementation

The skeleton code at `https://classroom.github.com/a/0wU5ck6t` is the start of this week's assignment. Write and test the following functions. Do not change the names and numbers of parameters (except when explicitly directed to), but feel free to add helper functions or specify types in function signatures. Use the test cases supplied with the assignment to check your solutions. Feel free to write more tests to better understand the program.

Do not use any of F#'s built-in functions for this assignment. For example, there is already a `Set.union` function, but you should implement your own for the sake of practice and understanding.

1.2.1 Goals

1. Practice writing tail-recursive F# functions
2. Practice using F# custom types

1.2.2 union

Using `isMember` from an earlier assignment, implement the `unionList` function. For the `unionList` function, assume that both input lists are sets and therefore have unique elements. The resulting list should also have unique elements.

Finish implementing the `union` operation on the `SET` data types `SI` and `SIIS`. The following `union` handles three cases and uses the `unionList` function.

```
let union s1 s2 =
  match (s1, s2) with
    | (I l1, I l2) -> I (unionList l1 l2)
    | (IS l1, IS l2) -> IS (unionList l1 l2)
    | (SISI l1, SISI l2) -> SISI (unionList l1 l2);;
```

1.2.3 difference

Set difference, $R \setminus S$, is all elements in R but not in S. For example:

```
> let R = IS [(3333, "COMP"); (3333, "HIST"); (3333, "PHYS")];;
> let S = IS [(3333, "COMP"); (3333, "BIOL")];;

> difference R S;;
val it : SET = IS [(3333, "HIST"); (3333, "PHYS")]
```

Write the `diffList` function to perform the difference operation with two lists.

Define the `difference` operation and test with `SI` and `SIIS` sets.

1.2.4 myRev

Give a tail recursive reverse function of $O(n)$ time for the type definition of `mylist` (which holds values of generic type `'element`):

```
type 'element mylist = NIL | CONS of 'element * 'element mylist;;
```

This type is a functional syntax for the infix `::` operator: `CONS(1,NIL)` is equivalent to `1::[]`. Write a helper function and `myRev` to work specifically with the `mylist` type.

Test with:

```
> let c = CONS (1, CONS (2, CONS (3, NIL)));;
> myRev c;;    // returns int mylist = CONS (3, CONS (2, CONS (1, NIL)))
```

1.2.5 vecadd

The below version of `vecadd` must calculate the recursive values of `vecadd` before cons'ing onto '(h1+h2)'. A tail recursive version would cons onto a known output value for each recursion step, passing down the accumulated value as an argument and returning the final output in the base case. This would have a different signature than `vecadd`, so you should write a helper function or local function to encapsulate this modification. Give a **tail recursive** version of the `vecadd` function with the same function signature as the one below.

```
let rec vecadd v1 v2 =
  match (v1, v2) with
    | ([], []) -> []
    | (h1::t1, h2::t2) -> (h1+h2)::vecadd t1 t2;;
```

The signature for your helper should have the signature: `val helper : int list -> int list -> int list -> int list`

Test with: `vecadd [1;2;3] [4;5;6]`

Note that `vecadd` should call your tail recursive helper version of `vecadd` function.

But wait! The results will be reversed. We can use the $O(n)$ `rev` function to re-reverse the output. Fix `vecadd` by reversing the result.

Give a runtime complexity estimate of the `vecadd` function that includes `rev`, based on the number of elements in the vectors.

1.2.6 mergeSort

Modify `mergeSort` to use an additional parameter of type `('a -> 'a) -> bool`. Use this function in place of the $x < y$ comparison to determine how to sort the input – whether it's ascending or descending or some other ordering. Assume that the input list is not empty.

For example:

```
mergeSort (fun x y -> x<y) [4;2;6;5];;         // returns int list = [2;4;5;6]

mergeSort (fun x y -> x>y) [4;2;6;5];;         // returns int list = [6;5;4;2]
```

Test your revised `mergeSort` with both ascending and descending sorts. In addition to ints, the updated `mergeSort` should work with other types, including sorting strings by length:

```
mergeSort (fun (a:string) (b:string) -> a.Length < b.Length) ["hi"; "hello"; "h"
    ];;
// returns string list = ["h"; "hi"; "hello"]
```

1.2.7 mylistMergeSort

Modify `mergeSort` to sort a `mylist` of any type on which relations > and < are defined. Use the name **mylistMergeSort** for this function. For example, sorting in descending order:

```
mylistMergeSort (fun x y -> x>y) (CONS (6, CONS (12, CONS (3, NIL))));;
// returns int mylist = CONS(12,CONS(6,CONS(3,NIL)))
```

2 Testing

Unit testing is supplied for this assignment. **Tests will not compile by default – you must change the signature of `mergeSort` in order to fix this.**

These are not exhaustive tests and you should try running the functions with other valid arguments as well.

Before submitting the assignment, make sure that you can run the tests. If you have any uncompleted functions, comment them out so that the tests can run on the parts that you have completed.

All of your function signatures (the parameter and return type) must be correct in order for the tests to run. Do not include any ';;' statement endings in your source code.

Mac directions:

1. Go to **View→Tests** and a panel should open
2. Click the 'Run all tests' button and you should get unit-by-unit feedback on test results.

Windows directions:

1. Go to the **Solution explorer** window.
2. Right-click on the top line in the window: **Solution 'comp3350_a1'** file→**Build Solution**.
3. Go to **Test→Run All Tests**.
4. The test names should appear in the **Test Explorer** window, and you can run them by clicking **Run All** in the **Test Explorer** window.

3 Grading

Be professional. Make results easy to understand and grade. Include only those parts to be graded, as well as their helper functions. Leave comments where necessary, especially if it aids in grading.

Each of the 6 implemented sections is worth $\frac{1}{6}$ of the assignment grade.