COMP 212 Fall 2022
Lab 09

1 Introduction

In this lab, you will experiement with the module system. In lecture, we discussed the module system as a way to clearly mark and enforce abstraction boundaries. In this lab, you will use modules from the perspective of both a client and an implementer.

One of the functions you will write returns an *option*, which are the following datatype:

```
datatype 'a option =
     NONE
   | SOME of 'a
```

That is, an option is like a boolean, except in one case it carries some data. For example, a function `int -> int option` can return —NONE— to signal that the function "failed" (didn't want to return a number), and might return —SOME 7— to "succeed" in produding the value 7.

2 Dictionaries

A *dictionary* is a datastructure that acts as a finite map from *keys* and to *values*. We represent a dictionary by a type

```
('k, 'v) dict
```

`dict` is a type constructor that takes two type arguments (unlike e.g. `'a list`, which takes only one). The first, `'k`, represents the type of keys, whereas the second, `'v` represents the type of values. For example, an `(int,string) dict` maps integers to strings.

There are many possible implementations of dictionaries, including using lists, trees, and functions. Since there are so many different ways to implement dictionaries, it would be nice if we could have an abstract interface to them that is the same regardless of the underlying implementation. This is where the module system comes in.

In `LabDict.sig`, we have provided the following signature for you to implement for dictionaries (see Figure 1).

The type and values in it have the following specifications:

```
signature LABDICT =
sig

  (* We model a dictionary as a set of key-value pairs written k ~ v:
     (k1 ~ v1, k2 ~ v2, ...) *)
  type ('k, 'v) dict

  (* the empty mapping *)
  val empty : ('k, 'v) dict

  (* insert (cmp, (k1 ~ v1, ..., kn ~ vn), (k,v) )
        == (k1 ~ v1, ..., ki ~ v,...) if cmp(k,ki) ==> EQUAL for some ki
     or == (k1 ~ v1, ..., kn ~ vn, k ~ v) otherwise
     *)
  val insert : ('k * 'k -> order) * ('k, 'v) dict * ('k * 'v) -> ('k, 'v) dict

  (* lookup (cmp, (k1 ~ v1,...,kn ~ vn), k) == SOME vi
                                               if cmp(k,ki) ==> EQUAL for some ki
                                            == NONE otherwise
     *)
  val lookup : ('k * 'k -> order) * ('k, 'v) dict * 'k -> 'v option

  val toString : ('k * 'v -> string) * ('k, 'v) dict -> string

end
```

Figure 1: Dictionary Signature

- `('k, 'v) dict` is an abstract type representing the type of the dictionary. Note that it is parametrized over two different types—`'k`, the type of keys, and `'v`, the type of values.
- `empty` is a dictionary that contains no mappings.
- `insert` is a function that takes a comparison function for keys, a dictionary, and a key-value pair and returns the dictionary with the mapping added. If the key is already in the dictionary, the new value supersedes the old one.
- `lookup` is a function that takes a comparison function, a dictionary, and a key, and returns `SOME v` if that key maps to the value `v` in the dictionary, or `NONE` if there is no mapping from the key.
- `toString` is a function that takes a way to show a key-value pair and displays a dictionary as a string — you can use this for testing

We have called this signature `LABDICT` because it is a version of dictionaries that is small enough for you to implement in lab; a real dictionary library would provide more operations.

3 Implementation: Binary Search Trees

First, you will implement dictionaries using a binary search tree (BST). Recall the discussion of binary search trees from when we implemented mergesort on trees: the key invariant is that, for every `Node(l,x,r)`, everything in `l` is less than or equal to `x`, and everything in `r` is greater than or equal to `x`.

To implement dictionaries, we will store both a key and a value at each node, using the following datatype:

```
datatype ('k, 'v) tree =
    Empty
  | Node of ('k, 'v) tree * ('k * 'v) * ('k, 'v) tree
```

The type `('k,'v) tree` is parameterized by two type variables, `'k` for keys and `'v` for values. For example, an `(int,string) tree` represents a dictionary mapping integers to strings. Note that we write `('k,'v) tree` with a comma separating the two type parameters, while in the `Node` constructor we write `'k * 'v` for the type of key-value pairs. In `Node(l,(k,v),r)`, `k` is the key and `v` is the value. Every key in `l` should be less than or equal to `k`, and every key in `r` should be greater than `k`. That is, the keys are sorted in the order we discussed earlier in the class; the values are just along for the ride.

Task 3.1 In the file `TreeDict.sml`, implement a structure `TreeDict` matching the signature `LABDICT`. You should use the datatype above as the internal representation of a dictionary.

To test your implementation, you can run the command

```
- CM.make "sources.cm";
```

from the REPL.

To test your code from the REPL, you will need to refer to functions inside your `TreeDict` structure as components of the module. (i.e. as `TreeDict.<function_name>` where `<function_name>` is the name of the function you want to run). Recall that you can only refer to functions that have been defined in the signature.

Note about the compilation manager: If you compile your code using `CM.make`, the compilation manager will compile all of the files specified in the `.cm` file. So you should use `CM.make` every time you want to compile.

Have us check your code for insert and lookup after writing each function!

4 Client: Word counts

Now you will write some client code in the module `Count` in `count.sml`.

We use a `(string,int) TreeDict.dict` sorted according to `String.compare` to map words to numbers.

Task 4.1 Write a function

```
val increment : (string,int) TreeDict.dict
                * string
                -> (string,int) TreeDict.dict
```

such that `increment (d, w)` increments the number associated with the word `w` if `w` occurs in `d`, or maps `w` to `1` if it does not already occur.

Task 4.2 Why should or shouldn't `increment` be in the client code, as opposed to the implementation of dictionaries?

Task 4.3 Write a function

```
count : string list
       * (string,int) TreeDict.dict
       -> (string,int) TreeDict.dict
```

The input to `count` is a list of strings and a dictionary mapping words to the frequencies with which they have appeared. The function should return an updated dictionary, which includes the counts for all of the words in any string in the input. Use the provided `words` function to divide a string into a list of strings, where each string in the result is a single word.

Have us check your code!