CS51 **Spring 2024**

Code Review 4: Modular Programming

1 Practice Problems

Problem 1.1 (Inverting a Stack)**.** Consider the polymorphic `Stack` module as shown in Listing 6.

1. Define a function `invert_stack : 'a Stack.stack -> 'a Stack.stack` that inverts the elements of the stack.

   Solution
   Suppose we consider this attempt at a solution:

   ```
   let rec invert_stack (s: 'a Stack.stack) : 'a Stack.stack =
     if s = Stack.empty then Stack.empty else
     let elt = Stack.top s in
     let rest = Stack.pop s in
     Stack.push elt (invert_stack rest) ;;
   ```

   This is incorrect. To see why, consider the recursion framework outlined in section 2.2 of the Code Review 1 notes. Even though the base case seems reasonable (returning an empty stack when the stack is empty), let's look at the recursive step. Again assume `invert_stack` works for small instances of the problem. Consider $s = [3;2;1]$. On the first recursive call, `elt` $= 3$ and `rest` $= [2;1]$. `invert_stack rest` should return the stack $[1;2]$. Pushing 3 onto the stack $[1;2]$ would give us $[3;1;2]$ which is different from the desired output of $[1;2;3]$. In fact, the entire recursion fails, and this version of `invert_stack` returns $[3;2;1]$, the same as the input stack.

   Now, let us consider another approach. Suppose we use an *accumulator* that we will add to on each recursive call.

   ```
   let invert_stack (s: 'a Stack.stack) : 'a Stack.stack =
     let rec invert_stack_aux (s : 'a Stack.stack) (acc : 'a Stack.
        stack) =
     if s = Stack.empty then acc else
       let elt = Stack.top s in
       let rest = Stack.pop s in
     invert_stack_aux rest (Stack.push elt acc) in
   invert_stack_aux s Stack.empty ;;
   ```

   For the base case, we return the accumulator `acc` when the stack is empty. In the recursive step, we update the accumulator and pass the remaining stack as `rest`. When `rest` reaches the empty stack, we then return the final accumulated value.

2. Write a few unit tests outside the `Stack` module testing the functionality of `invert_stack`.

Solution
Note that this the unit tests below are not exhaustive, but the solution provides a template for what you could do.

```
open CS51Utils ;;
open Absbook ;;

open Stack ;;

(* define a helper function to convert a stack to list *)
let rec stack_to_list (s: 'a Stack.stack) : 'a list =
  if s = empty then [] else
    let elt = top s in
    let rest = pop s in
    elt :: stack_to_list rest ;;

(* Define some example stacks. *)
let stack1 = empty |> push 5 |> push 4 |> push 3 ;;

let invert_stack_tests () =
  unit_test (invert_stack empty = empty) "invert_stack empty";
  unit_test (stack_to_list (invert_stack stack1) = [5; 4; 3]) "
     invert_stack stack of len 3" ;;

invert_stack_tests() ;;
```

Problem 1.2 (Graph). An (undirected) graph is a defined set of nodes and a set of edges, where each edge is a pair of different nodes.

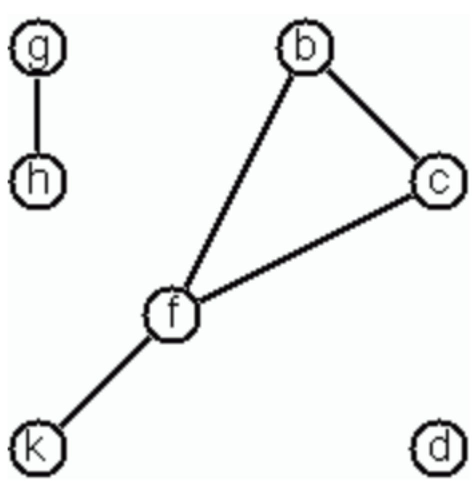

Figure 1: Graph Example

There are many ways to represent a graph. For this problem, we will consider the **adjacency list** representation, in which we represent a finite graph as a collection of unordered lists such that each list represents the neighbors (vertices that share an edge) with a particular vertex. For instance, we can represent the graph in Figure 1 as: `[('g', ['h']); ('h', ['g']); ('b', ['c', 'f']); ('c', ['b', 'f']); ('f', ['b', 'c', 'k']); ('k', ['f']); ('d', []) ].`

Here, each element in the list is a pair, where the first element is a vertex and the second element is a list of neighbors.

We will work on defining a `Graph` module.

First, consider the following signatures for a `Vertex` and `Graph` module.

Listing 1: Vertex Module Signature

```
module type VERTEX =
  sig
    type t
    val compare : t -> t -> int
  end ;;
```

Listing 2: Graph Module Signature

```
module type GRAPH =
sig
  type v
  type graph
  exception VertexAlreadyExists
  exception VertexDoesNotExist
  exception EdgeAlreadyExists

  (* empty graph *)
  val empty: graph

  (* add a vertex to the graph *)
  val addVertex : v -> graph -> graph

  (* adds an edge between two vertices `u` and `v` *)
  val addEdge : v -> v -> graph -> graph

  (* Return a list of the vertices in the graph` *)
  val vertices : graph -> v list

  (* Return the neighbors of a vertex `v` *)
  val neighbors : v -> graph -> v list

  (* Return the number of vertices and edges in the graph *)
  val graph_size : graph -> int * int
end ;;
```

1. We will define a functor `MakeGraph` that takes in a module of type `VERTEX` and returns a module of type `GRAPH`. What is the signature of this functor?

2. To define `MakeGraph`, start defining `type graph` based on the example given for Figure 1.

3. There are two main problems with this definition for `graph`. First, the `type` allows us to add the same vertex twice to the neighbors list for any vertex. Second, our list can contain the same vertex twice: for instance, the following is expressible `[('a', []); ('a', [])]`.

   Let's address the first problem by representing the neighbors as a **set**, which is an unordered collection of unique elements. Use the OCaml Set Module.

4. To address the second problem, suppose we instead represent the adjacency list using a **hash map** (or hash table). A hash map is a dictionary data structure in which we map unique keys to values. To do this, we'll use the OCaml Map Module, taking the keys to be the vertex and the values to be the neighbors set, as defined in part 3.

5. Define `addVertex` and `addEdge`. If the user attempts to add a vertex or edge that already exists, return the apporpriate exception.

6. Complete the implementation of the module by defining `vertices`, `neighbors`, and `graph_size`.

Problem 1.3 (Find Paths)**.**

1. Using the MakeGraph functor, define a module `IntGraph` such that the type of the vertices in the graph are integers.

Problem 1.4 (Set)**.** Let's implement the `Set` module from scratch. Sets are an unordered list of elements with no duplicates.

1. Define a module signature for an `ORDERED_TYPE` which specifies the type of the elements, a function to compare elements, and a function to convert each of the elements into strings.

2. Create a module type for `Set` that includes the following values and operations on sets are supportable.

   (a) `empty` : the empty set

   (b) `add` : add an element to a set

   (c) `take` : a function that takes an element in the set and returns the rest of the elements in the set as a pair : (h, t), where h is the extract element and t are the rest of the elements.

   (d) `mem` : check if an element is a member of a set

   (e) `union` : return the union of two sets

   (f) `intersection` : return the intersection of two sets

   (g) `print_set` : convert a set into a string.

3. Write a functor `MakeSet` that takes in a module of `ORDERED_TYPE` and returns a module of type `SET`.

4. Use `MakeSet` to define the following modules:

   (a) A set of integers

(b) A set of strings

(c) A set of int lists

(d) A set of integer sets

5. Define a function `power_set` that returns a set of all the subsets of the original set. For this problem, you can create a function that returns the power set for a set of integers. For instance, the power set of $\{1,2,3,4\}$ is

```
{{}, {1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4},
   {3, 4}, {1, 2, 3}, {1, 3, 4}, {1, 2, 4}, {2, 3, 4},  {1, 2, 3,
   4}}
```