CSCI 2041: Advanced Programming Principles
Sample Problems for Second Midterm Exam, Spring 2015

Included below are a few problems that should give you an idea of the kinds of questions you might encounter in the second mid term exam for CSci 2041. In the actual exam, there would four to seven questions of these kinds, drawn from the topics we would have covered up to the end of the lecture on April 3; while the focus will be on the material discussed since the first mid-term, you will still need to know things we talked about earlier. As you can see, the problems below are taken from the lectures, homeworks and labs. This is not surprising, the main purpose of the sample questions is to remind you of things we have studied so that you are sufficiently prepared with the concepts to work with them quickly in the exam.

Remember that the exam is a closed book one. You should therefore know pretty much everything that we have discussed in class to be able to work without crutches on problems that use the concepts. Small errors in syntax in OCaml programs that you write will be overlooked, but you should not make mistakes on conceptual issues.

Also note that the problems in an exam are typically of graded levels of difficulty. For this reason, it is important not to get stuck on any question for too long. Answer questions that you are sure about and that you can get done quickly first, and then return to the ones that will take more time.

Problem I.

For each of the functions below

- explicitly follow the process that was explained in connection with the definition of append in class to determine if the function definition is type correct, and
- at the end of it, indicate what type is inferred for the function.

You must show your work for the first of the items above for credit in this problem. Also, do not forget that you have to complete the type checking process to conclude that the function is in fact type correct; if you leave it half way through, it is not clear that the type you infer will actually work, and so you will lose credit.

1.
```
let compose f g = (fun x -> (f (g x)))
```

2.
```
let rec reduce f lst u =
     match lst with
     | [] -> u
     | (h::t) -> f h (reduce f t u)
```

3.
```
let rec append l =
    match l with
    | [] -> (fun l2 -> l2)
    | (h::t) ->
        let tail_appender = append t in
      (fun l -> h :: tail_appender l)
```

Problem II.

Define the following functions from first principles, i.e. without using any library functions.

1. The function

   ```
   map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
   ```

   that is like `map`, except that it maps the given function simultaneously over two lists to produce the new list. Here is an example interaction:

   ```
   # map2 (fun x y -> x + y) [1;2;3] [4;5;6];;
   - : int list = [5; 7; 9]
   #
   ```

2. The function

   ```
   divide_list : ('a -> bool) -> 'a list -> 'a list * 'a list
   ```

   that takes a boolean function over a given type and a list of elements of that type and divides the list into two lists, one containing all the elements that satisfy the boolean function and the other containing those that do not satisfy it. Here are some example interactions that characterize the desired behaviour of the function:

   ```
   # divide_list (fun x -> true) [];;
   - : 'a list * 'a list = ([], [])
   # divide_list (fun x -> true) ["a"; "string"; "list"];;
   - : string list * string list = (["a"; "string"; "list"], [])
   # divide_list (fun x -> false) ["a"; "string"; "list"];;
   - : string list * string list = ([], ["a"; "string"; "list"])
   # divide_list  (fun x -> if (x mod 2 = 0) then true else false)
                  [1;3;6;8;9;10];;
   - : int list * int list = ([6; 8; 10], [1; 3; 9])
   #
   ```

3. Given the following type definition

   ```
   type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
   ```

   the function

   ```
   treemap : 'a btree -> ('a -> 'b) -> 'b btree
   ```

   that takes a binary tree and a function that acts on the elements of the binary tree as input and transforms the tree into a new tree that is obtained by applying the function to each element in the tree.

Problem III.

Recall the definitions of the functions `reduce` and `accumulate`:

```
let rec reduce f lst u =
   match lst with
   | [] -> u
   | (h::t) -> f h (reduce f t u)

let rec accumulate f lst u =
   match lst with
   | [] -> u
   | (h::t) -> accumulate f t (f h u)
```

We want to use the ability these functions give us to iterate over lists to define other functions. Your task is to fill in the blanks below with OCaml expressions to yield definitions of the relevant functions.

1. `let append l1 l2 = reduce ___ l1 ___`
2. `let filter f l1 = reduce ___ l1 ___`
3. `let unzip l = reduce ___ l ___`
4. `let union l1 l2 = accumulate ___ l1 ___`; `union` is supposed to correspond to the unioon operation on two sets, where sets are represented by lists in which an element appears at most once.

Problem IV.

Transform the following functions into a tail-recursive form using continuations in the manner discussed in class. Note that you must follow that approach to receive any credit, just writing a tail-recursive definition will not do.

1. ```
   let rec sumlist lst =
       match lst with
       | [] -> 0
       | (h::t) -> h + (sumlist t)
   ```
2. ```
   let rec append l1 l2 =
          match l1 with
          | [] -> l2
          | (h::t) -> h::append t l2
   ```
3. Assuming the following representation of binary trees

   ```
   type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
   ```

   the following function for summing up the elements in the tree:

```
let rec sumTree t =
  match t with
  | Empty -> 0
  | Node (i,l,r) -> i + sumTree l + sumTree r
```

Problem V.

In homework 2, we used the following definition of the binary tree type

```
type ('a,'b) btree = Empty | Node of 'a * 'b * ('a,'b) btree * ('a,'b) btree
```

and we implemented the operations `find`, `insert` and `delete` for looking up, inserting, and deleting an item from a binary search tree. Later, we understood that there was a problem with not packaging the ordering and equality relations with the binary tree that was supposed to be a binary search tree. This led to the following type declaration:

```
type ('a,'b) bstree =
   { data : ('a,'b) btree;
     eq : 'a -> 'a -> bool; lss : 'a -> 'a -> bool;}
```

We defined a better version of the function `find` with the type

```
find : ('a,'b) bstree -> 'a -> bool
```

that did not need to rely on externally defined ordering and equality relations. (Actually, we did not explicitly represent data in addition to a key in what we did in class, but this is a simple extension.)

1. Following the example of the `find` function, define an improved version of the `insert` function with the type

   ```
   insert : ('a,'b) bstree -> 'a -> 'b -> ('a,'b) bstree
   ```

2. Do a similar thing with the `delete` function, redefining it to have the type

   ```
   delete : ('a,'b) bstree -> 'a -> ('a,'b) bstree
   ```

Problem VI.

1. The repeat-until construct from Pascal has the following syntax:

   ```
   repeat <statement> until <condition>;
   ```

   where `<statement>` can be any statement in Pascal and `condition` is any boolean-valued expression in Pascal. The meaning of this construct is that we should execute the statement, then repeatedly execute the loop so long as the condition remains false.

   Assuming the model for encoding statements in Pascal that we considered in class and in Lab 9, provide a definition for the identifier `repeatstat` so that it encodes this repeat-until construct.

2. Using the encoding of repeat-until described above, provide an encoding of the following program fragment from C that calculates the sum of the numbers up to a given positive number n:

```
i = 0;
sum = 0;
repeat  i = i + 1; sum = sum + i;
until (i >= n);
```

You should do this through the following steps:

(a) Explain how you will represent state as relevant to this program fragment and then define a type state corresponding to this representation.

(b) Based on your representation of state write the `get` and `put` functions for the variables used in this program fragment.

(c) Using a let declaration, bind the identifier `sumup` to an expression in OCaml that represents the shown program fragment.

Problem VII.

Consider the following OCaml function that calculates the sum of the natural numbers up to a given number:

```
let sumup n =
    let rec sumup_aux n acc =
        if (n = 0) then acc
        else sumup_aux (n-1) (n + acc)
    in sumup_aux n 0
```

Prove the correctness of this function by carrying out the following steps:

1. State a property of `sumup` that expresses its correctness, given the intended purpose of the function.

2. State a property of `sumup_aux` that can be proved and that, when proved, can be used to yield a simple proof of the correctness of `sumup`.

3. Using natural number induction, prove that `sumup_aux` has the property that you have stated for it.

4. Using the property you have proved for `sumup_aux`, argue that `sumup` must satisfy the property you have stated for it.

Problem VIII.

Recall the following functions on lists:

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | (h::t) -> h::(append t l2)

let rec sumlist lst =
  match lst with
  | [] -> 0
  | (h::t) -> h + (sumlist t)

let rec reverse l =
  match l with
  | [] -> []
  | (h::t) -> append (reverse t) [h]
```

Using induction on lists, prove the following fact about these functions:

$$\forall l \in (int\ list). \forall x \in int.\ (sumup\ (append\ l\ [x])) = sumup\ (x :: l)$$

$$\forall l \in (int\ list).\ sumup\ (reverse\ l) = sumup\ l$$

Once you have proved the first, you may use it in proving the second.

Problem IX.

Recall the following definitions pertaining to binary trees:

```
type 'a btree =
    Empty
  | Node of 'a * 'a btree * 'a btree

let rec sumTree t =
  match t with
  | Empty -> 0
  | Node (i,l,r) -> i + sumTree l + sumTree r

let rec insert t i =
  match t with
  | Empty -> Node (i,Empty,Empty)
  | Node (i',l,r) ->
      if (i < i') then Node (i',insert l i,r)
      else Node (i',l,insert r i)
```

Using induction on trees, prove the following fact about these functions:

$$\forall t \in (int\ btree). \forall i \in int.\ sumTree\ (insert\ t\ i) = (sumTree\ t) + i.$$