CSCI 2041: Advanced Programming Principles
Sample Problems for the Final 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 final exam for CSci 2041. In the actual exam, there would six to eight questions of these kinds, drawn from the topics we would have covered during the term. Remember that this is a cumulative exam: you can be asked questions about anything we have seen during the term. For this reason, you might want to look at the sample questions for the two mid terms as well. Of course, there will be a greater emphasis on the material we have covered since the second mid-term since you have not yet been tested on your understanding of that material. You might also note that some of 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. Note that how well you know the material determines how quickly you can work with it; thus, an exam could appear to be "too long" if you don't know the material well enough. Also note that small errors in syntax in OCaml programs that you write will be overlooked, but you should not make mistakes on conceptual issues. Finally, remember 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 let declarations below

- indicate whether or not OCaml will accept it without signalling an error
- if OCaml will indicate an error, explain what that error is, and
- if OCaml will accept it without errors, then present the value the identifier will be bound to and also its type; if the value is a function, explain briefly but precisely what the function does.

Obviously, the second and third possibilities above are mutually exclusive.

1.
```
let f b a = a + b in let g = f 8 in g 8
```

2.
```
let x = 4 in
    try (x + 2) / (x - 3) with
    | _ -> "Error! Division by 0"
```

3.
```
let x = let f y = let g z = y * z in g
          in (fun x -> (f 2 x) + (f 3 x))
```

4.
```
let rec addN =
    function
    | (_,[]) -> []
    | (n,(h::t)) -> (h+n) :: addN n t
```

5.
```
let y = let x = 5 in x + 3 in x + y
```

Problem II.

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

2.
```
let rec append l =
   match l with
   | [] -> (fun l2 -> l2)
   | (h::t) ->
       let tail_appender = append t in
     (fun l -> h :: tail_appender l)
```

3.
```
let rec zip lpair =
   match lpair with
   | (([],_) | (_,[])) -> []
   | ((h1::t1),(h2::t2)) -> (h1,h2) :: zip (t1,t2)
```

Problem III.

Transform the following functions into a tail recursive form by adding one more argument to the function that is a continuation that represents the computation that remains to be done after the recursive call has been completed.

1. 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
   ```

2. ```
   let revapp l1 l2 =
     match l1 with
     | [] -> l2
     | (h::t) -> revapp t (h::l2)
   ```

   Note that the transformed version of this function *must* take a continuation to satisfy the requirements of this problem.

3. The map function defined as

   ```
   let map f l =
    match l with
    | [] -> []
    | (h::t) -> (f h) :: map f t
   ```

   Assume that the function argument to the transformed version of `map` will itself have been transformed to take a continuation. Also assume that in the expression `e1 :: e2`, we must evaluate `e1` first and then `e2`.

Problem IV.

Consider the following familiar functions for appending two lists and for reversing a list

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

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

1. Write down a recurrence relation for the number of function calls that `append` will make as a function of the length of its argument lists.

2. Solve the recurrence relation you have written in the previous part. You can do this by expanding the recurrence and providing an analysis of the expansion that leads to a closed form expression or by positing a solution and confirming that it is correct using induction. One of the two forms of justifications must accompany your answer for credit.

3. Write down a recurrence relation for the number of function calls that `reverse` will make as a function of the length of its input argument.

4. As in part 2, use the recurrence relation for `reverse` to get a closed form expression for the number of function calls.

5. Is it reasonable to use a count of the function calls as a measure of the time taken by `append` and `reverse`? You must explain your answer for credit.

6. Is it reasonable to use the lengths of the lists as a measure of the sizes of the inputs in the assessment of the time complexity of `append` and `reverse`? You must explain your answer for credit.

Problem V.

1. How much new space will be consumed by the expression `(h :: t)` that is used in a function at a point where `h` and `t` are known values? Express your answer in terms of the sizes of `h` and `t`. Also, you must explain your answer for credit.

2. Where will the space considered in part 1 be allocated? In the heap or in the stack?

3. Develop recurrence equations for the heap and stack space taken by the `append` function in Problem IV as a function of the lengths of its list arguments and solve this recurrence equations to get an estimate of its space usage.

4. Develop recurrence equations for the heap and stack space taken by the function `reverse` in Problem IV as a function of the length of its list argument and solve this recurrence equations to get an estimate of its space usage. A question you must consider carefully in this assessment is how the space consumed by the recursive calls made to itself by `reverse` should be taken into account in this overall calculation. You must justify the choice you make.

Problem VI.

Consider the function

```
partition : ('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:

```
# partition (fun x -> true) [];;
- : 'a list * 'a list = ([], [])
# partition (fun x -> if (String.length x < 4) then true else false)
         ["this"; "is"; "a"; "list"; "of"; "mixed"; "length"; "strings"];;
  - : string list * string list =
(["is"; "a"; "of"], ["this"; "list"; "mixed"; "length"; "strings"])
# partition (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])
#
```

1. Provide a definition for this function from first principles, i.e., without using any predefined functions. Of course, you can use constructors such as `::` and `(_,_)` (for constructing pairs) in your definition.

2. Now assume that we are given the function `reduce` that is defined as follows:

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

   Provide an alternative definition for `partition` that uses this function by filling in the blank spaces in the let declaration shown below. The expressions you provide must not use any predefined functions.

   ```
   let partition p l = reduce ___ l ___
   ```

Problem VII.

Consider the function

```
unzip : ('a * 'b) list -> 'a list * 'b list
```

that takes a list of pairs and breaks it up into two lists, one containing the first element in each pair and the other containing the second element of each pair. Some example interactions illustrating the expected behaviour of this function are shown below:

```
# unzip [];;
- : 'a list * 'b list = ([], [])
# unzip [(1,2);(3,4);(5,6)];;
- : int list * int list = ([1; 3; 5], [2; 4; 6])
# unzip [("a","b");("c","d");("e","f")];;
- : string list * string list = (["a"; "c"; "e"], ["b"; "d"; "f"])
#
```

1. Provide a definition for this function from first principles, i.e., without using any predefined functions. Of course, you can use constructors such as `::` and `(_,_)` (for constructing pairs) in your definition.

2. Now assume that we are given the function `reduce` that is defined as follows:

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

   Provide an alternative definition for `unzip` that uses this function by filling in the blank spaces in the let declaration shown below. The expressions you provide must not use any predefined functions.

   ```
   let unzip l = reduce ___ l ___
   ```

Problem VIII.

Suppose that we have the following type declaration

```
type expr =
  | Const of int
  | Add of expr * expr
  | Mult of expr * expr
```

Present the induction principle associated with the `expr` type based on the definition.

Problem VI.

Consider the following function definitions:

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

let rec append l1 l2 =
  match l1 with
```

```
  | [] -> l2
  | (h::t) -> h::(append t l2)

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

Assume that the following property relating *sumlist* and *append* has already been proved:

$$\forall l_1\, \forall l_2.\, (sumlist\ l_1) + (sumlist\ l_2) = sumlist\ (append\ l_1\ l_2).$$

Note that l_1 and l_2 are of list type in the property written above. Also recall that when we use $=$ in writing properties such as these, we mean that the left and right sides have the same termination behaviour and that they evaluate to the same value if they terminate. The goal in this problem is to prove the following property holds:

$$\forall l_1\, \forall l_2.\, (sumlist\ (rev\_aux\ l_1\ l_2)) = (sumlist\ l_1) + (sumlist\ l_2).$$

Obviously, this property will have to be proved by induction.

1. Which of the lists l_1 and l_2 would be the right one to do induction on?
2. Let us refer to the list you have picked as l. In an inductive proof, you then have to show that a property of the form $P(l)$ holds for all values of l. Indicate what the property $P(l)$ is for which you will have to construct such an argument.
3. Based on your choices in the first two parts, indicate the property that you will have to prove for the base case.
4. Provide a proof for the base case. Remember, you must justify each step you take clearly and correctly, otherwise it is not a proof.
5. Now indicate what you will have to do in the inductive step. Specifically, present clearly the induction hypothesis and then present what you have to show given the induction hypothesis so as to complete the proof.
6. Provide an argument for the inductive step. Note again that you must justify each step in your argument clearly and correctly for it to constitute a proof.

Problem IX.

Consider the following function definitions:

```
let rec zip lpair =
  match lpair with
  | (([],_) | (_,[])) -> []
  | ((h1::t1),(h2::t2)) -> (h1,h2) :: zip (t1,t2)
```

```
let rec unzip l =
  match l with
  | [] -> ([],[])
  | ((h1,h2)::t) ->  let (t1,t2) = unzip t in (h1::t1,h2::t2)
```

The goal in this problem is to prove the following property by induction:

$$\forall l.\, (zip\ (unzip\ l)) = l$$

Note that l is of list type in the property shown above.

1. What would you do induction on to show the above property holds?
2. Let us refer to the item you have picked in the first step as x. In an inductive proof, you then have to show that a property of the form $P(x)$ holds for all values of x. Indicate what the property $P(x)$ is for which you will have to construct such an argument.
3. Based on your choices in the first two parts, indicate the property that you will have to prove for the base case.
4. Provide a proof for the base case. Remember, you must justify each step you take clearly and correctly, otherwise it is not a proof.
5. Now indicate what you will have to do in the inductive step. Specifically, present clearly the induction hypothesis and then present what you have to show given the induction hypothesis so as to complete the proof.
6. Provide an argument for the inductive step. Note again that you must justify each step in your argument clearly and correctly for it to constitute a proof.

Problem X.

Consider the following function definitions

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

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

let plus x y = x + y
```

The goal in this problem is to provide an inductive proof for the following property:

$$\forall l.\, sumlist\ l = (reduce\ plus\ l\ 0).$$

Note that l is of list type in the property shown above.

1. What would you do induction on to show the above property holds?

2. Let us refer to the item you have picked in the first step as x. In an inductive proof, you then have to show that a property of the form $P(x)$ holds for all values of x. Indicate what the property $P(x)$ is for which you will have to construct such an argument.

3. Based on your choices in the first two parts, indicate the property that you will have to prove as the base case.

4. Provide a proof for the base case. Remember, you must justify each step you take clearly and correctly, otherwise it is not a proof.

5. Now indicate what you will have to do in the inductive step. Specifically, present clearly the induction hypothesis and then present what you have to show given the induction hypothesis so as to complete the proof.

6. Provide an argument for the inductive step. Note again that you must justify each step in your argument clearly and correctly for it to constitute a proof.

Problem XI.

Note: In an exam you may expect to have to write at most a couple of the kinds of functions asked for in the problem. I am clubbing them all together so as to save myself some effort in putting together this sample questions set.

Recall the following declarations that identify a stream type and functions for creating and probing streams:

```
type 'a stream = Stream of (unit -> 'a * 'a stream)
let mkStream f = Stream f
let nextStream (Stream f) = f ()
```

We have seen a few examples of how to create streams using these definitions. One example we saw defined the stream of natural numbers starting from 1:

```
let rec fromNStream n = mkStream (fun () -> (n, fromNStream (n+1)))
let natStream = (fromNStream 1)
```

We also saw a definition of the function `take` for forming a list of a specified number of initial elements from a given stream:

```
let rec take n s =
   match n with
   |  0 -> []
   |  n ->
        let (x,rst) = nextStream s in
          (x :: take (n-1) rst)
```

1. Define the stream `fibStream : int stream` that corresponds to the sequence of fibonnaci numbers. An example interaction that demonstrates the use of this stream:

```
# take 7 fibStream;;
- : int list = [1; 1; 2; 3; 5; 8; 13]
#
```

2. Define the function

   ```
   mapStream : ('a -> 'b) -> 'a stream -> 'b stream
   ```

   that generates a stream that corresponds to applying a given function to every element in a given stream. An interaction that demonstrates the use of this function:

   ```
   # let squareStream = mapStream (fun x -> x * x) natStream;;
   val squareStream : int stream = Stream <fun>
   # take 5 squareStream;;
   - : int list = [1; 4; 9; 16; 25]
   #
   ```

   Note that `squareStream` above is itself a stream, i.e. `mapStream` generates an object whose items will be produced one at a time, only on demand.

3. Define the function

   ```
   zipStreams : 'a stream -> 'b stream -> ('a * 'b) stream
   ```

   that takes two streams and creates a stream of pairs of corresponding elements from the given streams. An interaction that demonstrates the use of this function:

   ```
   # let natfibStream = zipStreams natStream fibStream;;
   val natfibStream : (int * int) stream = Stream <fun>
   # take 5 natfibStream;;
   - : (int * int) list = [(1, 1); (2, 1); (3, 2); (4, 3); (5, 5)]
   # let natsquareStream = zipStreams natStream squareStream;;
   val natsquareStream : (int * int) stream = Stream <fun>
   # take 5 natsquareStream;;
   - : (int * int) list = [(1, 1); (2, 4); (3, 9); (4, 16); (5, 25)]
   #
   ```

4. Define the function

   ```
   unzipStream : ('a * 'b) stream -> 'a stream * 'b stream
   ```

   that takes a stream of pairs and produces from it two separate streams defined by the first and the second elements respectively of each pair. An example interaction that demonstrates the use of this function:

```
# let (s1,s2) = unzipStream natsquareStream;;
val s1 : int stream = Stream <fun>
val s2 : int stream = Stream <fun>
# take 5 s1;;
- : int list = [1; 2; 3; 4; 5]
# take 5 s2;;
- : int list = [1; 4; 9; 16; 25]
#
```

Observe again that `unzipStream` generates a pair of *streams*, i.e. the items in these streams will be generated only on demand.

Problem XII.

Consider the following function definitions

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

let sqr n = n * n

let rec take l n =
  match (l,n) with
  | ((_,0) | ([],_)) -> []
  | (h::t,n) -> h :: (take t (n-1))
```

1. Show the step-by-step evaluation of the expression

   ```
   take (map sqr [1;2;3]) 2
   ```

   under *call-by-name* evaluation. You *must* show each step in the sequence so as to make clear how exactly computation proceeds. Note also that when a list is shown at the interactive level, each element of the list is evaluated.

2. Repeat the above but this time using *call-by-value* evaluation.

Problem XIII.

[**Note:** In an exam, you would be asked to do one or the other of the two parts that appear below in a problem of this sort. I am clubbing the two parts into one problem to save some writing effort.]

The *subset-sum* problem is the following: given a set of positive numbers and another number, to find a subset of the given set that sums up to the given number. No "good" algorithm is currently known to solve this problem; it is what is known as an NP-complete problem. In this problem you are to write a program to solve subset-sum using search.

Specifically, we will represent sets of numbers by lists in which any given number appears at most once. Then the overall task is to define the function

```
subset_sum : int list -> int -> int list
```

However, to get credit for this problem, you must define this function using search and implementing this search in the particular ways described below.

1. The search should be structured as follows: the function does a recursion over the list argument that represents the set, deciding at each point whether or not to include the head item in the sublist being constructed. If the constructed sublist adds up to the given number, then we have a solution. Otherwise, we have to backtrack to the most recent choice and try a different alternative. **For credit for this part, the search that is described must be realized using exceptions.**

2. The search in this part is structured as before. **However, for credit for this part, the search must be realized by using two continuations:** a success continuation that is called when success is encountered and a failure continuation that, in effect, encapsulates the action to be taken in case of failure.

Problem XIV.

[**Note:** In an exam, you would be asked to do one or the other of the two parts that appear below in a problem of this sort. I am clubbing the two parts into one problem to save some writing effort.]

This problem involves finding a path between a source and a destination node in an undirected graph. This is a problem for which we do know "good" algorithms: one of these algorithms is due to Dijkstra, a (deceased) computer scientist who also played a major role in the formative period of work on programming and programming languages. While there are better ways to solve this problem, we explore a simple but inefficient way to do it that uses search.

We will assume that graphs are given by adjacency lists, i.e., by a list of pairs where the first item in each pair represents a node and the second item is a list of all the nodes that are adjacent to it. Thus, the following represents a graph that has 1, 2, 3, and 4 as its nodes and in which 1 is adjacent to 2 and 3 and 2 is additionally adjacent to 3:

```
[(1,[2;3]) ; (2,[1;3]) ; (3,[1;2])]
```

The task is to then define the function

```
find_path : 'a -> 'a -> ('a * 'a list) list -> ('a * 'a) list
```

that takes a source node, a destination node and an adjacency list of the kind described above and that returns a list of edges constituting a path from the source to the destination if such a path exists and the empty list otherwise. Some example interactions that demonstrate the behaviour expected of this function:

```
# find_path 1 3 [(1,[2;3]) ; (2,[1;3]) ; (3,[1;2])];;
- : (int * int) list = [(1, 2); (2, 3)]
# find_path 1 4 [(1,[2;3]) ; (2,[1;3]) ; (3,[1;2])];;
- : (int * int) list = []
#
```

To get credit for this problem, the function must use search and also particular ways of realizing search as described below.

1. The function should be structured as follows. The source node and the adjacency list are used to determine the candidates for the next node in the path. These candidates are then tried in sequence: one of them becomes the new source node in the recursion and the remaining "neighbours" are held over in case failure is encountered. To realize backtracking in this part, you must use exceptions in the way we have discussed in class.

2. The search in this case must be structured the same way as in the first part. However, the implementation of the backtracking search must be realized through two continuations, one to call if success is encountered and the second to call to realize backtracking action in the case of failure.

Note: Since we are dealing with undirected graphs, you have to be careful to avoid getting into cycles because then your function will not terminate. Towards this end, keep a list of "seen" nodes and avoid using them as the "next" nodes in any path to be explored.

Problem XV.

Recall the code that we used initially in class to realize delayed evaluation:

```
type 'a delay = Delay of (unit -> 'a)
let mkLazy e = Delay e
let force (Delay e) = e ()
```

In this problem, we will consider encapsulating this code via the modules feature in OCaml.

1. Consider the following signature in OCaml:

   ```
   module type DELAY =
   sig
     type 'a delay
     val mkLazy : (unit -> 'a) -> 'a delay
     val force : ('a delay) -> 'a
   end
   ```

   Using the code shown at the beginning of this problem, define a module called `Delay` that satisfies the requirements imposed by the `DELAY` signature and whose internal structure is not visible from the outside, i.e. its external view is completely determined by the `DELAY` signature.

2. Consider the following code

   ```
   let x = Delay (fun () -> 3 + 4) in let Delay y = x in y ()
   ```

   This code assumes that the delay type is visible externally. As such, it will not work if the implementation of delays is provided only by means of the `Delay` module defined in the first part. Fix the code so that it will work in this case.

3. Is there a reason why we might prefer the encapsulated view of the delay type that is provided by the `Delay` module to the open view that is provided by making the code directly visible? Be brief but still specific in your answer.

   Hint: consider changing the implementation of delays from the naive version to the form we considered in class where we evaluated the expression lazily but did it only once