CSCI 2041: Advanced Programming Principles
Sample Problems for First 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 first mid term exam for CSci 2041. In the actual exam, there would six or seven questions of this kind, covering the range of topics that I outlined in the lecture on Feb 20 as things that we have covered in the last four and a half weeks. You have encountered problems like these in the homeworks, in class discussions and in the labs. If you have worked on the labs and homeworks diligently and have also attended lectures regularly, you should be well prepared for the exam. Nevertheless, work through these problems so as to know what you should expect in the exam and also to be sure you can answer such questions quickly; speed can be important in exams that run for limited time like this one does.

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.

Problem 1.

For each of the expressions below, indicate if they are well-typed. If they are well-typed, indicate their values and their types. If they are not well-typed, explain why not.

1.
   ```
   match [1;2] with
   | [] -> "string"
   | (h::t) -> h
   ```
2. `let f x = x ^ " world" in let y = "hello" in (f y, y)`

3.
   ```
   match [(1,"string")] with
   | [] -> "Not found"
   | ((y,v)::t) -> if (1= y) then v else t
   ```
4. `if (3 = 2) then 7`
5. `if (3 + 2) then 5 else 7`

Problem 2

For each of the expressions below, indicate whether or not OCaml will deem it to be legal. If an expression is legal, present its type and its value. If it is not legal, explain what the problem is. Assume that there are no let bindings preceding these expressions, i.e., they are presented to OCaml in an empty context.

1. `let x = 1 in x + y`
2. `let x = 1 in let y = x + 1 in x + y`

3. `let x = 1 in let x = 5 and y = x + 1 in x + y`

4. `let f y = (let x = 5 in y * x) + y in f 7`

5.
```
  let f y =
     let z = (let x = 5 in y * x) + y in "hello" in (f 7) ^ " world"
```

Problem 3

For each of the function definitions below

- explicitly follow the process that was explained in connection with the definition of `append` in class to determine if the following function definitions are type correct, and
- at the end of it, either conclude that the definition is not type correct or show the type that would be inferred for the function.

Note that you must show your work to get any credit in this problem.

1.
```
let rec pairwith x l =
    match l with
    | [] -> []
    | (h::t) -> (x,h) :: pairwith x t
```

2.
```
let rec reverse l =
    match l with
    | [] -> []
    | (h::t) -> (reverse t) :: h
```

3.
```
let rec zip lp =
    match lp with
    | ([],_) -> []
    | (_,[]) -> []
    | ((x::l1),(y::l2)) -> (x,y) :: zip (l1,l2)
```

4.
```
let rec unzip =
    function
    | [] -> ([],[])
    | ((f,s)::l) -> let (fl,sl) = unzip l in (f::fl,s::sl)
```

Problem 4

Consider the following iterative program that computes a somewhat strange but still well-defined value for each given value for a positive number `n0`:

```
n = n0;
v = 1;
while (n != 0) {
  if (n mod 2 = 0)
  then { v = 2 * v + (v * v); n = n / 2; }
  else { v = 3 * v; n = n - 1; }
}
return v;
```

1. How many arguments will the corresponding tail recursive version of this function have?
2. Translate the iterative code into its tail recursive form on OCaml. Call the function you define `f'`.
3. Use the function `f'` to define an OCaml function `f` that takes `n0` as its argument and returns the value computed for it by the iterative program.

Problem 5

Recall the type we saw in class for representing data that is either an integer or a string:

```
type intorstr = Int of int | Str of string
```

Recall also the type we defined in class for representing binary trees:

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

Define a function `sumTree : (intorstr btree) -> int` that sums up all the integer values that appear in a given `intorstr` binary tree. The following interaction exemplifies the behaviour we want of `sumTree`:

```
# let tree1 = Node (Str "foo",
                  Node(Int 5, Node (Str "bar",
                                    Empty,
                                    Node (Int 7,Empty,Empty)),
                              Node (Int 3, Empty, Empty)),
                  Empty);;
          val tree1 : intorstr btree =
  Node (Str "foo",
   Node (Int 5, Node (Str "bar", Empty, Node (Int 7, Empty, Empty)),
    Node (Int 3, Empty, Empty)),
   Empty)
# sumTree tree1;;
- : int = 15
#
```

Problem 6

Consider the following pruned down version of the `expr` type we defined in class for representing a small subset of OCaml expressions

```
type  expr =
  Id of string | Int of int
| Plus of expr * expr | Minus of expr * expr
```

We often write programs that have expressions that can be simplified before we actually try to run them. For example, using the representation described by the above type, if we have an expression of the form

```
   Plus (Int 3, Int 7)
```

we can recognize this when processing the expression and replace it immediately by `(Int 10)`. This kind of simplification can actually be done recursively. For example, if we see an expression of the form

```
  Minus (Int 5, Plus (Int 3, Int 7))
```

we would first think of simplifying the inner `Plus` expression and then this would yield an outer `Minus` expression that can be simplified. Carrying out this simplification, we can replace the outer expression by `(Int -5)`.

Define a function `simplify : expr -> expr` that carries out the kind of simplification described above. The following interactions exemplify the behaviour we desire of `simplify`:

```
# simplify (Plus (Int 3, Int 7));;
- : expr = Int 10
# simplify (Minus (Int 5, Plus (Int 3, Int 7)));;
- : expr = Int (-5)
# simplify (Plus (Minus (Int 5, Plus (Int 3, Int 7)), Id "x"));;
- : expr = Plus (Int (-5), Id "x")
#
```

Problem 7

In this problem, we will assume that the types in OCaml are determined by a small, fixed set of basic types and type constructors. In particular, we will assume that something is a type in OCaml only if it can be obtained in one of the following ways:

- it is `int` or `bool`,
- it is a type variable, given simply by a string denoting its name,
- it is the type of lists of elements of type `ty` for some type `ty`, or
- it is the type of functions from some type `ty1` to some type `ty2`.

Define a type `ocamlTy` in OCaml for representing these types as data.

Problem 8

Lists that mix elements of a certain type with lists of elements of that type are sometimes used in programming in languages that do not have rich data structuring abilities like OCaml. Consider, for example, a tree that we might represent in OCaml as follows

```
Node (5,
      Node (2,Node (1,Empty,Empty),
              Node (3,Empty,Node(4,Empty,Empty))),
      Node (7, Node (6,Empty,Empty),
               Node (9,Node (8, Empty, Empty),Empty)))
```

Since Scheme has no data structures other than lists, such a tree would have to be represented by a list that looks something like this:

```
[5 ; [2 ; [1; []; []] ; [3; []; []]];
     [7 ; [6; []; []]; [9; [8; []; []]; []]]]
```

Unfortunately, the "list" we have written down above is not a good onw in OCaml because it has items of different types in it; some are integers, and some are lists that again have mixed kinds of items.

1. Define a one-place type constructor `nestedlist` in OCaml with suitable value constructors such that objects of type `(ty nestedlist)` represent nested lists over the type `ty`. To be specific, a nested list over a type `ty` is a list whose items are either of type `ty` or, recursively, nested lists over type `ty`. As a concrete example, the list that is shown above is a nested list over the type `int`.

2. Define a function `flatten: 'a nestedlist -> 'a list` that takes a nested list and produces from it a simple list of all the elements of type `'a` that appear in it. For example, given a representation of the nested list shown above, this function should produce the list `[5; 2; 1; 3; 7; 6; 9; 8]` as a result.