CS450: Progrmg Languages & Compilers
Fall 2014

Homework 2

University of Illinois, Urbana-Champaign,

Due: June 13, 2015, 5.00 pm. Submit PDF + code files on Moodle.

1 Theoretical Questions (40 points)

Objectives and Background

The purpose of this HW is to test your understanding of

- the scope of variables, and the state of environments used during evaluation;
- how an application of a function is evaluated;
- the order of evaluation in OCaml

Another purpose of HWs is to provide you with experience answering non-programming written questions of the kind you may experience on the midterms and final.

What to Turn In

Your answers to the following questions are to be submitted electronically via the Moodle. You should type up your answers to each of the problems below and submit them in a file called `hw2.pdf`. You can use LaTeX or any other wordprocessor of your choice to type up the solutions but we can only grade PDF files. (You can even write your answers by hand legibly. You should then make a reasonable effort to make a readable scan of it, and submit those documents as PDF file; please don't submit raw image files.)

We shall only grade PDF submissions. We will ask you to resubmit solutions in any other file format.

Problems

Problem 1: (24 points)

Below is a fragment of OCaml code, with various program points indicated by numbers with comments.

```
let p1 = (1., 3.);;
let p2 = (2., 5.);;
(* 1 *)
let slope (x,y) = y /. x;;
(* 2 *)
let sub (x1,y1) (x2,y2) = (x2 - x1, y2 - y1);;
(* 3 *)
let slope p1 p2 = slope (sub p1 p2);;
(* 4 *)
```

```
let slope_p2 = slope p2;;
(* 5 *)
let p2 = (3., 9.);;
(* 6 *)
slope_p2 p1;;
(* 7 *)
slope p1 p2;;
(* 8 *)
```

- (12 points) For each of program points 1, 2, 3, 4, 5 and 6, please describe the environment in effect after evaluation has reached that point. You may assume that the evaluation begins in an empty environment, and that the environment is cumulative thereafter. The program points are supposed to indicate points at which all complete preceding declarations (including local ones) have been fully evaluated. In describing the environments 1 through 5, you may use set notation, as done in class, or you may use the update operator `+`. If you use set notation, no duplicate bindings should occur. The answer for program point 6 should be written out fully in set notation without the update operator or duplications.
- (4 points) Show step by step how (* 5 *) is evaluated.
- (4 points) Show step by step how `slope_p2 p1` is evaluated.
- (4 points) Show step by step how `slope p1 p2` is evaluated.

Problem 2: (16 points)

Below is a fragment of OCaml code.

```
let f g x =
  (let r =
    if ((print_string "a"; x > 5) && (g(); x > 10))
    then
       (print_string "b"; x - 7)
    else
       let z = (print_string "c"; 15) in (print_string "d"; z)
    in (g(); r));;
let u = (f (fun () -> print_string "e\n") (f (fun () -> print_string "f\n") 3));;
```

Describe everything that is displayed on the screen after this code is pasted into an interactive OCaml session. This should include both the type information that the compiler gives back for each declaration, and any other things printed to the screen. Give explanations for all the other things printed, and the order in which they are printed. For the type information, no explanation is required but it should be correct.

2 Machine Problems (40 points)

Objectives and Background

The purpose of this MP is to help students master

- pattern matching;
- recursion; and
- lists

Another purpose of MPs in general is to provide a framework to study for the exam. Several of the questions on the exam will appear similar to the MP problems. By the time of the exam, your goal is to be able to solve any of the following problems with pen and paper in less than 2 minutes.

What to Turn In

You should put code answering each of the problems below in a file called `mp2.ml`. A good way to start is to copy `mp2-skeleton.ml` to `mp2.ml` and edit the copy. Please read the Guide for Doing MPs.

Problems

Note: In the problems below, you do not have to begin your definitions in a manner identical to the sample code, which is present solely for guiding you better. However, you have to use the indicated name for your functions and values, and they will have to conform to any type information supplied, and have to yield the same results as any sample executions given, as well as satisfying the specification given in English.

Problem 1: (2 point)

Write a function `rev_apply` which takes a function `f` and a pair `(x,y)`, interchanges `x` and `y` and applies the function `f` to both.

```
# let rev_apply f (x, y) = ...;;
val rev_apply : ('a -> 'b) -> 'a * 'a -> 'b * 'b = <fun>
# rev_apply (fun n -> n + 1) (2, 3);;
- : int * int = (4, 3)
```

Problem 2: (4 points)

Consider the following mathematical definition of a sequence s_n:

$$s_n = \begin{cases} 1 & \text{if } n \leq 1 \\ 3 * s_{\frac{n}{2}} & \text{if } n \text{ is even} \\ 2 + s_{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Write a recursive function `s` that takes an integer n and returns an integer s_n.

```
# let s n = ...;;
val s : int -> int = <fun>
# s 9;;
- : int = 29
```

Problem 3: (5 points)

Run-length encoding (RLE) is a data compression technique in which maximal (non-empty) consecutive occurrences of a value are replaced by pa pair of the value and a counter showing how many times the value was repeated in that consecutive sequence. For example, RLE would encode the list `[1;1;1;2;2;2;3;1;1;1]` as: `[(1,3);(2,3);(3,1);(1;3)]`. Write a function `rle` that takes a list and encodes it using the RLE technique. The function is required to use (only) forward recursion (no other form of recursion). You may not use any library functions.

```
# let rec rle lst = ... ;;
val rle : 'a list -> ('a * int) list = <fun>
# rle [1;1;1;2;2;2;3;1;1;1];;
- : (int * int) list = [(1, 3); (2, 3); (3, 1); (1, 3)]
# rle ['a';'b';'a';'a';'a';'c'];;
- : (char * int) list = [('a', 1); ('b', 1); ('a', 3); ('c', 1)]
```

Problem 4: (5 points)

Write a function `merge` that takes two lists and returns a list. Assuming that the two input lists are sorted, the result should be a sorted list containing elements from both the lists.

```
# let rec merge l1 l2 = ...;;
val merge : 'a list -> 'a list -> 'a list = <fun>
# merge [1;2;5;6] [3;4;6;9];;
- : int list = [1; 2; 3; 4; 5; 6; 6; 9]
```

Problem 5: (4 points)

Write a function `separate` that takes a list of integers and outputs a pair of lists with the first list containing all the odd integers in the original list and the second list contains all the even integers of the original list. The order of the integers in both lists must be the same as the original list.

```
# let rec separate l = ...;;
val separate : int list -> int list * int list = <fun>
# separate [1; 3; 2; 4; 5];;
- : int list * int list = ([1; 3; 5], [2; 4])
```

Problem 6: (6 points)

Write a function `maxsumseq` that takes a list of integers and outputs the maximum sum of the elements in any subsequence. [*Hint 1*: To obtain maximum sum subsequence, a global sum can be kept for the list and updated if current sum is greater than the global sum at any point. The current sum can be reset to 0 when it is less than 0.] [*Hint 2*: You can define local (recursive, perhaps) function(s)]

```
# let maxsumseq l = ...;;
val maxsumseq : int list -> int = <fun>
# maxsumseq [-1; 3; 2; -2; 5; -16];;
- : int = 8
```

Problem 7: (7 points)

Recall that a directed graph is a pair $G = (V, A)$ where

- V is a set of vertices or nodes,
- A is a set of ordered pairs of vertices, called arcs, directed edges, or arrows.

When we implements the directd graph, we usually use an initial interval of integers to represent V, and an adjacency matrix or list to represent the set A.

In this problem we will use an adjacency list to represent the set A. This means that we will use the integer list list to represent the adjacency list of a graph, and it has one list per node in the integer list list. The i^{th} position of the list contains all the nodes connected to the i^{th} node by an out-going edge, where the integers in each list represent the position of the corresponding node in the adjacency list. The labeling of nodes start from zero, and elements in the list are counted from zero. Write a function `check_adj` that takes an adjacency list and a pair of integers to check if there exists an edge from the first element to the second one. Remember that this is a directed graph. In this problem, you may use any library functions you choose. Also, the input integers in the lists will always be nonnegative, and you can safely assume that we will not test on any number which is bigger than the graph size minus one. However, if the input pair of integers contains a negative number, you should return false. Note OCaml's type inference may infer a more general type for your solution than the type listed below, so do not panic.

```
# let check_adj adj_list (a,b) = ...
val check_adj : 'a list list -> int * 'a -> bool = <fun>
# check_adj [[1;2;3;4];[3;0;4;5];[1;4;3;5];[2;1];[1;2];[2;3;4]] (0,3);;
- : bool = true
```

Problem 8: (7 points)

Write a function `cumsum` that takes a list of integers as input and returns a list of integers such that the ith element of the output list is the sum of all the elements from the beginning of the original list to its ith element. You must only use tail recursion for this problem. You may not use any library functions except `@`.

```
# let cumsum l = ... ;;
val cumsum : int list -> int list = <fun>
# cumsum [1;2;3];;
- : int list = [1; 3; 6]
```

3 Guide for Doing MPs

1. Setup a directory for CS 421 and a subdirectory for HWs/MPs.

2. Create a subsubdirectory corresponding to this HW within them.

3. Retrieve the directory for this MP posted on Moodle and all its contents. Get into that directory. (If we revise an assignment, you will need to repeat this to obtain the revision.)

4. To make sure you have all the necessary pieces, start by executing make. This will create the grader executable. Run the executable (`>$ ./grader`). Examine the failing test cases for places where errors were produced by your code. At this point, everything should compile, but the score will be 0.

5. Read and understand the problem for the handout on which you wish to begin working. (Usually, working from top to bottom makes most sense.) There is a tests file in this directory. This is an important file containing an incomplete set of test cases; you'll want to add more cases to test your code more thoroughly. Reread the problem from the handout, examining any sample output given. Open the tests file in the mpX directory. Find the test cases given for that problem. Add your own test cases by following the same pattern as of the existing test cases. Try to get a good coverage of your function's behaviour. You should even try to have enough cases to guarantee that you will catch any errors. (This is not always possible, but a desirable goal.) And yes, test cases should be written even before starting the implementation of your function. This is a good software development practice.

6. If necessary, reread the statement of the problem once more. Place your code for the solution in mpX.ml (or mpX.mll or mpX.mly as specified by the assignment instructions) replacing the stub found there for it. Implement your function. Try to do this in a step-wise fashion. When you think you have a solution (or enough of a part of one to compile and be worth testing), save you work and execute make and the ./grader again. Examine the passing and failing test cases again. Each failure is an instance where your code failed to give the right output for the given input, and you will need to examine your code to figure out why. When you are finished making a round of corrections, run make, followed by ./grader again. Continue until you find no more errors. Consider submitting your partial result so that you will at least get credit for what you have accomplished so far, in case something happens to interfere with your completing the rest of the assignment.

7. When your code no longer generates any errors for the problem on which you were working, return to steps 3) and 4) to proceed with the next problem you wish to solve, until there are no more problems to be solved.

8. When you have finished all problems (or given up and left the problem with the stub version originally provided), you will need to submit your file along with your PDF submission on Moodle.