
HW 1 – Evaluation and Environments

CS 421 – Fall 2008

Revision 1.1

Assigned September 16, 2008

Due September 23, 2008, 2:00 pm, in class

Extension 48 hours (20% penalty)

1 Change Log

1.1 Changed `strange` to `weird` in the last problem.

1.0 Initial Release.

2 Turn-In Procedure

Your answers to the following questions are to be hand-written, or printed, neatly on one or more sheets of paper, each with your name in the upper right corner. The homework is to be turned in in class at the start of class. Alternately, you may hand it to Prof. Elsa Gunter in person before the deadline.

3 Objectives and Background

The purpose of this HW is to test your understanding of

- the order of evaluation of expressions in OCaml;
- the scope of variables, and the state of environments used during evaluation

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.

4 Problems

1. (15 pts) Below is a fragment of OCaml code, with various program points indicated by numbers with comments.

```
let x = "a";;  
let y = 8;;  
(* 1 *)  
let f x z = x + y + z;;  
(* 2 *)  
let z =  
  let y = "b" in  
(* 3 *)  
    x ^ y;;  
(* 4 *)  
let g w = f w w;;  
let y = g y;;
```

(* 5 *)

For each of program points 1, 2, 3, 4 and 5, 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 have been fully evaluated. In describing the environments 1 through 4, 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 5 should be written out fully in set notation.

2. (20 pts) Below is a fragment of Ocaml code. Describe everything that is displayed on the screen (its observable behavior) after this code has been cut-and-pasted into an interactive Ocaml session, and explain why this behavior is observed. This should include both the type information that the compiler gives back for each declaration, and any other things printed to the screen. For the type information, no explanation is required (but it should be correct). Give explanation for all other things printed, and the order in which they occur.

```
let f g x =
  (let r =
    if (((print_string "a"; x > 2) && (print_string "b"; x > 10)) &&
      (let y = (print_string "c"; 14) in
        (print_string "d"; x > y) || (print_string "e"; x < 5)))
    then
      let z = (print_string "f"; 7) in (print_string "g"; x - z)
    else
      let z = (print_string "h"; 15) in (print_string "i"; z)
    in (g(); r));;
let u = (f (fun () -> print_string "j\n") (f (fun () -> print_string "k\n") 3));;
```

3. (24 pts) Consider the following body of Ocaml code:

```
(* 1 *) let double1 =
  (print_string "Calling double1 ... \n";
  (fun () -> fun x -> x * 2));;
(* 2 *) let double2 =
  fun () -> (print_string "Calling double2 ... \n";
  (fun x -> x * 2));;
(* 3 *) let double3 =
  fun () -> fun x ->
    (print_string "Calling double3 ... \n"; x * 2));;
(* 4 *) let result1 = let s = double1 () in List.map s [1;2;3];;
(* 5 *) let result2 = let s = double2 () in List.map s [1;2;3];;
(* 6 *) let result3 = let s = double3 () in List.map s [1;2;3];;
(* 7 *) let result4 = List.map (double1 ()) [1;2;3];;
(* 8 *) let result5 = List.map (double2 ()) [1;2;3];;
(* 9 *) let result6 = List.map (double3 ()) [1;2;3];;
(* 10 *) let result7 = List.map (fun x -> double1 () x) [1;2;3];;
(* 11 *) let result8 = List.map (fun x -> double2 () x) [1;2;3];;
(* 12 *) let result9 = List.map (fun x -> double3 () x) [1;2;3];;
```

Indicate what observable behavior (as described in the previous problem) will occur with the evaluation of each declaration. Explain why the evaluation of each declaration generates the described behavior. You may omit explanations of type information.

4. (Extra Credit) (5 pts) What is the running time of the following Ocaml function, in proportion to the size of the integers input? How can you rewrite this code so that its running time is linear in the size of the input?

```
let rec weird n =  
  if n <= 0 then 0  
  else if weird(n - 1) mod 2 = 0 then weird (n - 1) / 2  
  else weird (n - 1) - 1;;
```