HW 3 – Order of Evaluation

CS 421 – Fall 2013
Revision 1.0

Assigned Wednesday, September 11, 2013
Due Tuesday, September 17, 2013, 19:59pm
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this HW is to test your understanding of:

- Order of evaluation in OCaml

3 What to `handin`

Answer all parts of the problem below, save your work as a PDF (either scanned if handwritten or converted from a program), and hand in the PDF. The name of this assignment is `hw3`.

4 Problems

1. (20 pts) Below is a fragment of Ocaml code. Describe everything that is displayed on the screen (its observable behavior) after each top-level declaration in this code has been cut-and-pasted into an interactive Ocaml session, (followed by a carriage return) 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 explanations for all other things printed and the order in which they occur.

   Note: In Ocaml, in the application of an expression of function type to an argument, the arguament is evaluated to a value first, then the expression of function type is evaluated to a functional value. If the functional value is a closure (as opposed to a primitive operation, or a partial application of a primitive operation), then the resulting application of the closure to a value is done as described in class.

   For this problem, we expect, and will accept, an English narrative describing the sequence of computations and branch decisions (e.g. `x > 5` evaluates to `true` since `x` has a value of 12 and $12 > 5$, and therefore we evaluate the `then` branch).

```
let f = (print_string "\na\n";
         fun x ->
          (let r = (print_string "b"; x + 7)
           in if (print_string "c"; x > 0)
                 then (print_string "d\n"; 2 * x)
              else (print_string "e\n"; r)
          )
        );;
(* 1 *)

let g y = (print_string "z\n"; y + 2);;
(* 2 *)

let n = g(f(g 0));;
(* 3 *)
```

Solution:

1. The expression in the declaration for `f` is a sequence, and the first expression in that sequence is a `print_string "\na\n"`, so a newline, an `a`, and a second newline are printed.
2. The second expression in the sequence is a `fun` expression of type `int -> int`, which evaluates immediately to a closure. The identifier `f` is bound to that closure and the resultant binding information is displayed:

   ```
   val f : int -> int = <fun>
   ```

3. The next declaration binds `g` to the value of a function expression (starting with `fun y ->`). Agian this function expression evaluates immediately to a closure to which `g` is bound, and the resultant binding information is displayed:

   ```
   val g : int -> int = <fun>
   ```

4. To evaluate the expression for `n`, `g(f(g 0))`, we must begin by evaluating `f(g 0)`, and for that, we must evaluate `g 0`. Thus we evaluate the body of `g` with `0` as the value for `y`.
5. The body of `g` is a sequence, and so we begin by evaluating the first expression in the sequence, which is a print statement, and so a `z` and a newline are printed.
6. Next, the second expression in the sequence, `y + 2` where `y` is bound to `0`, is evaluated to return a value for `g 0` of `2`.
7. Having computed a value of `2` for the expression `g 0`, we now proceed to compute `f(g 0)`, which means computing the value of the closure for `f` with `2` as the value for its formal argument `x`.
8. The body of the closure for `f` begins with a `let_in_` statement, locally binding `r`. The expression giving the value for `r` is again a sequence with the first statement being a print statement. This prints `b`.
9. The second expression in the sequence calculates a value of `9` for `x + 7` where `x` is bound to `2`, and locally binds `r` to `9`.
10. Next we must evaluate the `in` part of the local binding. This is an `if_then_else_` expression, so we must begin by evaluating the boolean guard in an environment where `x` is bound to `2` and `r` is bound to `9`.
11. The boolean guard is a sequence, the first expression of which is a print statement, and so `c` is printed.
12. The second expression in the sequence is `x > 0`, which evaluates to `true` because `x` is bound to `2`.

13. The value of the second expression in the sequence is the value of the sequence, so the boolean guard evaluates to `true`.

14. Since the boolean guard evaluates to `true`, we evaluate just the `then` branch.

15. The then branch is a sequence, with the first expression being a print statement. This causes a `d` to be printed.

16. Then second expression `2 * x` evaluates to `4` because we are still in the scope of the binding of `x` to `2`.

17. This is the result of the `then` branch, and therefore the value of the whole `if_then_else_` expression, and hence the value of the `let_in_` expression, which is the body of the closure to which `f` is bound. Thus `f (g 0)` evaluates to `4`.

18. Having computed a value of `4` for the expression `f (g 0)`, we now proceed to compute `g (f (g 0))` by computing the body of the closure for `g` (again), this time with the formal parameter `y` bound to `4`.

19. The body of the closure for `g` is a sequence and the first expression in the sequence is a print statement, so, once again it prints a newline, an `a` and another newline.

20. Next we evaluate second expression, `y + 2` where `y` is bound to `4`, to get a value of `6`. This is the value of `g (f (g 0))`, and the value bound to `n`. The result of the binding then is printed:

    ```
    val g : int -> int = <fun>
    ```

The following is a transcript summarizing the above:

```
# let f = (print_string "\na\n";
           fun x ->
            (let r = (print_string "b"; x + 7)
             in if (print_string "c"; x > 0)
                  then (print_string "d\n"; 2 * x)
                else (print_string "e\n"; r)
            )
          );;

a
val f : int -> int = <fun>
# let g y = (print_string "z\n"; y + 2);;
val g : int -> int = <fun>
# let n = g(f(g 0));;
z
bcd
z
val n : int = 6
#
```