CS450: Progrmg Languages & Compilers
Fall 2014

Homework 1

University of Illinois, Urbana-Champaign,

Due: June 7, 2014, 5.00 pm. Submit PDF + code files on Moodle.

1 Theoretical Questions (15 points)

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.

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 `hw1.pdf`. You can use LaTeXor 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: (15 points)

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

```
let x = 5;;
let a = x * 4;;
(* 1 *)
let g x = x + (3 * a);;
(* 2 *)
let a = g (g 3);;
(* 3 *)
let b =
    let f x y = g x + g (y + a) in
(* 4 *)
f x (g a);;
(* 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 (including local ones) 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 without the update operator or duplications.

2 Machine Problems (15 points)

Objectives and Background

The purpose of this MP is to test the student's ability to

- start up and interact with OCaml;
- define a function;
- write code that conforms to the type specified (this includes understanding simple Ocaml types, including functional ones);

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 `mp1.ml`. A good way to start is to copy `mp1-skeleton.ml` to `mp1.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: (1 point)

Declare a variable `title` with the value `"MP 1 -- Basic OCaml"`. It should have type `string`. It shoud not contain a "newline".

Problem 2: (1 point)

Declare a variable `e` with a value of `2.71828`. It should have the type of `float`.

Problem 3: (2 points)

Write a function `firstFun` that returns the result of mulitplying a given integer by 2 and adding 5.

```
# let firstFun n = ... ;;
val firstFun : int -> int = <fun>
# firstFun 12;;
- : int = 29
```

Problem 4: (2 points)

Write a function `divide_e_by` that returns the result of dividing the value you gave in Problem 2 by the given `float`. You will not be tested on the value `0.0`.

```
# let divide_e_by x = ...;;
val divide_e_by : float -> float = <fun>
# divide_e_by e;;
- : float = 1.
```

(Your value may vary slightly from that printed here if you use a machine of different precision.)

Problem 5: (3 points)

Write a function `diff_square_9` that takes an integer and if the integer is between 3 and -3, squares it and substracts 9, and otherwise substracts its square from 9.

```
# let diff_square_9 m = ...;;
val diff_square_9 : int -> int = <fun>
# diff_square_9 5;;
- : int = -16
```

Problem 6: (3 points)

Write a function `dist_double` that, when given a string and an integer, first prints the string, a comma and a space, then "`I guess it's double or nothing!`" followed by a newline, and then returns double the integer it was given.

```
# let dist_double s n = ...;;
val dist_double : string -> int -> int = <fun>
# dist_double "Well, Sam" 8;;
Well, Sam, I guess it's double or nothing!
- : int = 16
```

Problem 7: (3 points)

Write a function `swizzle` that takes a 4-tuple (quadruple) and returns the 4-tuple with the first component moved to the third, the third moved to the second, the second moved to the fourth and the fourth moved to the first.

```
# let swizzle (w,x,y,z) = ... ;;
val swizzle : 'a * 'b * 'c * 'd -> 'd * 'c * 'a * 'b = <fun>
# swizzle (1,2,3,4);;
- : int * int * int * int = (4, 3, 1, 2)
```

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 the 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.