CS 4: Fundamentals Of Computer Programming, Winter 2017

Assignment 1: Evaluation and recursion

Due: *Friday, January 20, 02:00:00*

Coverage

This assignment covers the material up to lecture 4, corresponding to section 1.2.3 of SICP. Some material from recitation 1 is also covered.

What to hand in

All of your code should be saved to a file named `lab1.ml`. This file should be submitted to csman. Please indicate in a comment what problem any piece of code in your submission refers to. For instance:

```
(* A.2 *)
...
```

refers to problem 2 of section A.

Time hints

Each problem comes with a time hint as a number in bold text at the beginning of the description. This is our estimate of the maximum time that the problem should take you to solve (in minutes). If you find that a problem takes significantly longer, you should let us know so we can better calibrate the assignments. You should also talk to the TAs/lecturer to find out if the problem is that you just don't understand a concept, or that the problem is really unreasonably hard or long.

Part @: OCaml notes

Interface (`.mli`) files

For this and all subsequent assignments, we will be supplying you with an OCaml interface file to be used with your code. For this assignment, the file is called `lab1.mli` (note the `.mli` extension), and you should download it into the same directory that you are using to write and test your `lab1.ml` code.

The interface file consists mostly of type signatures of functions; occasionally it will have other things as well. The `lab1.mli` interface file looks like this:

```
(* Interface file for lab1.ml *)

val sum_of_squares_of_two_largest : int -> int -> int -> int
val factorial : int -> int
val e_term : int -> float
val e_approximation : int -> float
val is_even : int -> bool
val is_odd : int -> bool
val f_rec : int -> int
val f_iter : int -> int
val pascal_coefficient : int -> int -> int
```

The `val` declarations indicate that the type signature of a particular value is being described. Here, all such values are functions (functions are values in OCaml!). For instance, the `factorial` function has the type signature:

```
val factorial : int -> int
```

which indicates that it takes one argument (an `int`) and returns an `int`, as you would expect. Functions which take more arguments (like `pascal_coefficient`, which takes two integer arguments) have a somewhat less intuitive type signature:

```
val pascal_coefficient : int -> int -> int
```

You might have expected something like this instead:

```
val pascal_coefficient : int int -> int  (* WRONG *)
```

The reason why this is wrong is that arguments to OCaml functions are automatically *curried*, which means that they can be partially applied. (The name "curried" is a tribute to Haskell Curry, a logician who provided much of the theoretical foundations for modern functional programming languages.) In this case, it means that if we call `pascal_coefficient` with only one argument (an integer), it will return a function that takes the other integer argument and returns the integer result. Currying can occasionally give rise to confusing error messages but it's also extremely handy in practice, as we will see.

OK, let's assume you've written all of your code and you want to check that it conforms to the type declarations in the `.mli` file. How do you do that? The simplest way is to compile your code along with the interface file from the command line:

```
$ ocamlc -c lab1.mli lab1.ml
```

If no error messages are printed, your code is at least type-correct! Also, if you list the files in your directory, you will see two new ones: `lab1.cmi` and `lab1.cmo`. These are the (byte-code) compiled versions of the `lab1.mli` and `lab1.ml` files, respectively. Note that you have to put the `.mli` file before the `.ml` file in the command line; this won't work:

```
$ ocamlc -c lab1.ml lab1.mli
```

unless the `.cmi` file has already been compiled, in which case you don't have to have `lab1.mli` on the command line anyway. This is a bit annoying, but we live with it.

You can now load the `.cmo` file into an interactive OCaml session as follows:

```
# #load "lab1.cmo";;
```

In this case, nothing will be printed if there are no errors. In contrast, if you typed:

```
# #use "lab1.ml";;
```

then OCaml would print out the signature of every value in `lab1.ml`. We tend to use `#use` more than `#load` when interactively developing code, because `#use` will compile the code for you. You should know both forms.

Let's go back to what would happen if you typed:

```
# #load "lab1.cmo";;
```

You might expect that you could then use all the functions in `lab1.ml` (as you could if you'd used `#use`). Actually, that isn't the case (yet). If you try, this will happen:

```
# pascal_coefficient;;
Error: Unbound value pascal_coefficient
```

Huh? We just loaded `lab1.ml`, and `lab1.ml` defines `pascal_coefficient`, so why the error message? It turns out that `#load` loads the code as a separate *module* called `Lab1` (the name of the file, without the extension, and with the first letter capitalized). This is like saying `import lab1` in Python. We can get the function by using the module name as a prefix:

```
# Lab1.pascal_coefficient;;
- : int -> int -> int = <fun>
```

Notice that we have to capitalize the first letter of `Lab1`. Module names are always capitalized in OCaml.

If this is too tedious, you can dump all the names in the module into the local namespace by using an `open` declaration:

```
# open Lab1;;
# pascal_coefficient;;
- : int -> int -> int = <fun>
```

This is like saying `from lab1 import *` in Python. We will have much, much more to say about the OCaml module system in future assignments. OCaml actually has the most powerful module system of any computer language in wide use.

Once you have finished writing your code, you should always compile it against the `.mli` file we provide to check that your code not only compiles but has the type signature we want. (Using `#use` will check that your code compiles, but it may not have the type signature we want.)

Part A: Basic exercises

1. Below is a sequence of expressions. What is the result (the type and value or the error message) printed by the OCaml interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented. If the interpreter indicates an error, explain briefly (one sentence) why the error occurred. There are also some other questions below which you should answer to the best of your ability. Note that entering each code fragment interactively requires that you add the `;;` terminator to terminate input as described above. Write your answers in comments.

    1. `10`
    2. `10.`
    3. `5 + 3 + 4`
    4. `3.2 + 4.2`
    5. `3 +. 4`
    6. `3 + 4.2`

7. `3 +. 4.2`
8. `3.0 +. 4.2`
9. `9 - 3 - 1`
10. `9 - (3 - 1)`
11. `let a = 3`
12. `let b = a + 1`
13. `a = b`
14. `[1; 2; 3] = [1; 2; 3]`
15. `[1; 2; 3] == [1; 2; 3]` Is this the same as or different from the previous expression? Why?
16. `[1, 2, 3]` Explain why this gives the result it does. This is a nasty pitfall which highlights one of the less desirable features of OCaml's syntax.
17. `if b > a && b < a * b then b else a`
18. `if b > a and b < a * b then b else a`
19. `2 + if b > a then b else a`
20. `if b > a then b` This is not a syntax error. Why does this give a type error? *Hint*: What does OCaml assume if the `else` in an `if/then/else` form is left off?

2. [**15**] SICP, exercise 1.3

Define a function that takes three integer numbers as arguments and returns the sum of the squares of the two larger numbers. Call the function you define `sum_of_squares_of_two_largest`. You will probably find the `&&` special operator to be handy.

3. [**15**] SICP, exercise 1.4

Observe that our model of evaluation allows for combinations whose operators (functions) are compound expressions. Use this observation to describe the behavior of the following function:

```
let a_plus_abs_b a b =
  (if b > 0 then (+) else (-)) a b
```

Write your answer in a comment. Note that surrounding an operator with parentheses makes it into a two-argument function, so

```
(+) 2 3
```

is the same as:

```
2 + 3
```

Part B: Evaluation

In this section, write all essay-question-type answers inside OCaml comments.

1. [**25**] SICP, exercise 1.5

   Before tackling this problem, read the section in the book called "applicative order and normal order" (not covered in class!). Applicative-order evaluation is just the evaluation rule we described in class; it's sometimes called "strict evaluation". Normal-order evaluation is an alternative to the evaluation rule we learned in class. In normal-order evaluation, nothing is evaluated unless it needs to be to get the final result (it's sometimes called "call-by-need" where applicative-order evaluation is called "call-by-value"). Normal-order evaluation is used in some functional languages like Haskell (more accurately, Haskell uses *lazy evaluation*, which is a more efficient version of normal-order evaluation).

   Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two functions:

   ```
   let rec p () = p ()
   let test x y = if x = 0 then 0 else y
   ```

   Then he evaluates the expression:

   ```
   test 0 (p ())
   ```

   What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form `if` is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

   NOTE: This problem doesn't require a lengthy explanation; two or three sentences should be enough.

2. [**25**] SICP, exercise 1.6. This problem is one of my (Mike's) favorites.

   Alyssa P. Hacker doesn't see why `if` needs to be provided as a special syntactic form. "Why can't I just define `if` as an ordinary function?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if` using pattern matching:

   ```
   let new_if predicate then_clause else_clause =
     match predicate with
       | true  -> then_clause
       | false -> else_clause
   ```

   [Of course, since this is a function it will have to be called using function syntax (the `if` syntax is built-in to OCaml).] Eva demonstrates its use to Alyssa:

   ```
   # new_if (2 = 3) 0 5;;
   - : int = 5
   # new_if (1 = 1) 0 5;;
   - : int = 0
   ```

   Delighted, Alyssa uses `new_if` to write the following program to compute square roots:

   ```
   let square x = x *. x
   let average x y = (x +. y) /. 2.0

   let improve guess x = average guess (x /. guess)
   let is_good_enough guess x =
     abs_float (square guess -. x) < 0.00001

   let rec sqrt_iter guess x =
   ```

```
    new_if (is_good_enough guess x)
           guess
           (sqrt_iter (improve guess x) x)
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

3. [**60**] SICP, exercise 1.9

   Each of the following two functions defines a method for adding two positive integers in terms of the functions `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

```
let rec add_a a b =
  if a = 0
     then b
     else inc (add_a (dec a) b)

let rec add_b a b =
  if a = 0
     then b
     else add_b (dec a) (inc b)
```

   Note that `inc` or `dec` could trivially be defined as:

```
let inc a = a + 1
let dec a = a - 1
```

   but for this problem, assume that they are primitive functions.

   Using the substitution model, illustrate the process generated by each function in evaluating `add_a 2 5` and `add_b 2 5`. Are these processes iterative or recursive?

   For this problem only, we want you to write out the substitution model evaluation in great detail. That means you have to actually write out the fact that numbers evaluate to themselves, built-in functions evaluate to their internal representations, functions with arguments desugar to their corresponding `fun` forms, etc. Don't skip steps or you'll get a 1 on this problem (and hence, on the section and on the lab as a whole!). Also note when names are bound to their values.

   One shortcut that you can take is to replace the body of a function with ellipses (`...`), or to replace parts of it that aren't relevant with ellipses. Please indent your work to make it obvious when you are evaluating a subexpression, a sub-sub-expression, etc. Be explicit about writing out evaluate, apply, and substitution steps, and when you are desugaring a function definition into the equivalent `fun` form. Also note where you are invoking a special form rule distinct from the usual evaluation rule (*e.g.* with an `if` expression).

   Assume that function arguments evaluate from left to right. This isn't necessary to get the right result, but it will make it easier for your graders to grade if everyone does this the same way.

   Write the substitution model evaluation in an OCaml comment. Our solution for each part is around 60-70 lines long. If your solutions are much shorter than that then you are skipping too many steps.

   We realize that some of you (OK, *all* of you) may dislike this problem. Think of it the same way you might think of taking cod liver oil or eating broccoli; unpleasant but ultimately good for you. Understanding how a computer evaluates expressions is fundamental knowledge, and we will revisit this idea several times in this course (but it will never again be as tedious as this problem!).

Part C: Recursion

1. In this problem we're going to write a function that enables us to compute the number *e*, the base of natural logarithms, which is equal to 2.7182818... We will do this by summing a part of an infinite series expansion which computes e: e = 1/0! + 1/1! + 1/2! + ... where n! is the factorial of n (which in turn is n * (n-1) * (n-2) * ... * 1).

   We'll start out with the factorial function itself:

   ```
   (* This function computes the factorial of the input number,
      which for a number n is equal to n * (n-1) * ... * 1. *)
   let rec factorial n =
     if n = 0 then 1 else n * factorial (n - 1)
   ```

   Note: The definition of `factorial` has to be included in your lab submission or the rest of the code won't work!

   a. **[5]** Write a simple function called `e_term` which takes a (non-negative) integer argument and computes that term of the infinite series expansion of *e*. This function is not recursive. Note that the result must be a floating-point number; use the `float_of_int` function to convert from an OCaml `int` to a `float`. *Note*: OCaml never implicitly promotes one numeric type to another, so you must do it explicitly if that's what you want.

   b. **[20]** Write a recursive function called `e_approximation` that takes one positive integer argument and computes an approximation to *e* (an "e-proximation", as it were) by summing up that many terms of the infinite series expansion of *e* (actually, it'll sum up the first n+1 terms, since it starts at term 0 and ends at term n). Write the function as a linear recursive process. Use your `e_term` function to help you write this one.

   c. **[5]** Compute an approximation to *e* by summing up to the 20th term of the infinite series expansion. Write down the answer that OCaml gives you in a comment in your lab submission. Then write down the value of `exp 1.0` which is `e` to the power of `1`, and compare with the result given by `e_approximation 20` (they should be nearly identical).

   d. **[10]** What happens if you try to compute a better approximation to *e* by summing up to the 100th term of the infinite series expansion? Why does this happen? Write your answer in a comment.

2. [**20**] It's possible to have a kind of recursion which involves more than one function; this is called *mutual recursion*. A simple example is a pair of functions `is_even` and `is_odd`. `is_even` is a predicate which returns `true` if its argument is an even (non-negative) integer and `false` if its argument is an odd (non-negative) integer (zero is considered even). `is_odd` returns `true` if its numeric argument is odd and `false` otherwise. Write these functions, using only recursion, testing for equality with zero, and subtracting 1.

   Note that when defining mutually recursive functions in OCaml, you need to use the `let rec ... and ...` syntax:

   ```
   let rec f1 a b c = ...   (* ... may include f2 *)
   and f2 x y z = ...       (* ... may include f1 *)
   ```

   This works for any number of mutually-recursive functions. The whole form starts with a `let rec` and then each function after the first starts with `and`.

3. [**30**] SICP, exercise 1.11

   A function `f` is defined by the rule that `f(n) = n` if `n < 3` and `f(n) = f(n - 1) + 2f(n - 2) + 3f(n - 3)` if `n >= 3`. Write a function that computes `f` by means of a recursive process. Write a function that computes `f` by means of an iterative process.

Hint: The recursive definition is straightforward. Use it to check the correctness of the iterative definition. The iterative definition will need a helper function that keeps track of the last three numbers in the series, among other things.

4. [**30**] SICP, exercise 1.12

   The following pattern of numbers is called Pascal's triangle:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
   ...
```

   The numbers at the edges of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a function called `pascal_coefficient` which takes two `int` arguments (corresponding to the row number (starting from 1) and the index inside the row (also starting from 1)) and computes elements of Pascal's triangle by means of a recursive process.

   Use pattern matching to make the code cleaner. This is *not* optional! Use of `if`/`then`/`else` in your solution will result in a 0 for this problem.

   For arguments that don't correspond to locations in Pascal's triangle (like numbers < 1 or index numbers that are greater than the row number) you should signal an error using the code: `failwith "invalid arguments"`. This raises an exception. We'll talk more about exceptions later in the course.

   Examples:

```
# pascal_coefficient 1 1 ;; (* row 1, index 1 in row *)
- : int = 1
# pascal_coefficient 2 1 ;;
- : int = 1
# pascal_coefficient 2 2 ;;
- : int = 1
# pascal_coefficient 3 1 ;;
- : int = 1
# pascal_coefficient 3 2 ;;
- : int = 2
# pascal_coefficient 3 3 ;;
- : int = 1
# pascal_coefficient 10 5 ;;
- : int = 126
# pascal_coefficient 1 0 ;;
Exception: Failure "invalid arguments".
```

   Hints:

   - It's much easier to write this as a recursive process (*i.e.* with pending operations) than as an iterative process, though both approaches will involve recursive functions.

   - Pattern match on a tuple of both input arguments. Use the `when` form in a pattern when you need to specify non-structural conditions for a match (like two things being equal). Only use `when` when you can't use structural pattern matching, though. For instance, don't do this kind of thing:

```
match x with
  x' when x' = 1 -> ...
  ...
```

     when you can do this instead:

```
match x with
  1 -> ...
  ...
```

Also don't forget about the `_` (wildcard) syntax for don't-care patterns. Remember that a number in a pattern matches that literal number only.

Despite all this explanation, this function only needs to be a few lines long.

Copyright (c) 2017, California Institute of Technology. All rights reserved.