CS 4: Fundamentals Of Computer Programming, Winter 2017

Assignment 5: Tags and messages

Due: *Friday, February 24, 02:00:00*

Coverage

This assignment covers the material up to lecture 12, as well as the material in recitation lecture 3.

What to hand in

All of your code should be saved to a file named `lab5.ml`. This file should be submitted to csman as usual.

Part @: OCaml notes

Testing

For this assignment, we are supplying you with these support files:

1. a `.mli` OCaml interface file (for this assignment: `lab5.mli`)
2. a test script for parts A and B: `tests_lab5.ml`)
3. a test script for part C: `tests_lab5c.ml`)
4. a `Makefile`.)

You should download all of these files into the directory in which you are writing your `lab5.ml` code. You should not change them, and you should not submit them as part of your assignment submission to csman.

Once your assignment is done, you should compile it and check that it conforms to the interface file by entering this command:

```
$ make
```

Of course, you can also compile your code from inside the OCaml interpreter using the `#use` directive, as we've previously described. To run the test script for parts A and B, type this:

```
$ make test
```

This will compile the `tests_lab5.ml` file (which contains the unit tests) and output an executable program called `tests_lab5`. Then it will run that program and report all test failures.

If you want to compile the code and immediately run the part A/B test script (useful during development/debugging), type:

```
$ make all
```

Running the tests generates some log files; to get rid of them (as well as all compiled OCaml files), type:

```
$ make clean
```

The test script for part C is not automated, because your results may not be identical to the ones we show below (though they should be equivalent). Instead, it will display the result of evaluating various expressions, and you can check by eye that they are correct. To run the test script for part C, type this:

```
$ make testc
```

Or better still:

```
$ make testc | more
```

so you can see the results without having to scroll back in your terminal. Hit the spacebar to page through the results.

Note that most of the tests/examples described below are also in the test script. However, the test script doesn't test everything (particularly in part C), so don't assume that your code is perfect if it passes the test script.

Imperative programming and arrays

Part A of this assignment requires you to write some code in an imperative style, and to work with OCaml's arrays. Here is a quick refresher on imperative syntax in OCaml; see recitation lecture 3 and lecture 12 for more details. Also check out the OCaml manual here and this section and this chapter in Real World OCaml.

Imperative "variables" are modeled in OCaml using "reference cells" or "refs" for short. We create these with the `ref` function:

```
# ref 10;;
- : int ref = {contents = 10}
# let r = ref 42;;
val r : int ref = {contents = 42}
```

A reference is just a record with one mutable field called `contents`. You can access this field's value in two ways:

```
# r.contents;;  (* dot syntax for accessing record fields *)
- : int = 42
# !r;;
- : int = 42
```

The second way (using the `!` operator) is much shorter and is greatly preferred.

You assign to a ref cell in one of two ways:

```
# r.contents <- 1001;;  (* dot syntax for mutating record fields *)
- : unit = ()
# r;;
- : int ref = {contents = 1001}
# r := 2002;;
- : unit = ()
# r;;
- : int ref = {contents = 2002}
```

```
# !r;;
- : int = 2002
```

Again, using the `:=` operator is shorter and is greatly preferred.

Arrays are constructed using this notation:

```
# [| 1; 2; 3; 4; 5 |];;
- : int array = [|1; 2; 3; 4; 5|]
```

You can also use the `Array.make` function:

```
# Array.make 5 "foo";;
- : string array = [|"foo"; "foo"; "foo"; "foo"; "foo"|]
```

Array elements are fetched and changed using these syntaxes:

```
# let arr = Array.make 5 0;;
val arr : int array = [|0; 0; 0; 0; 0|]
# arr.(0);;
- : int = 0
# arr.(0) <- 42;;
- : unit = ()
# arr;;
- : int array = [|42; 0; 0; 0; 0|]
```

Note that arrays are always mutable, so any element in an array can be changed with the syntax given above.

OCaml has imperative `while` and `for` loops. Examples:

```
# let r = ref 10;;
val r : int ref = {contents = 10}
# while !r > 0 do
    Printf.printf "%d\n" !r;
    r := !r - 1
  done;;
10
9
8
7
6
5
4
3
2
1
- : unit = ()

# for i = 1 to 10 do
    Printf.printf "%d\n" i
  done;;
1
2
3
4
5
6
7
8
9
10
- : unit = ()
```

```
# for i = 10 downto 1 do
  Printf.printf "%d\n" i
  done;;
10
9
8
7
6
5
4
3
2
1
- : unit = ()
```

The Array module has a number of other useful functions on arrays (*e.g.* `Array.length`).

That should be enough to get you through this assignment.

Part A: Mutation and imperative programming

In this section, we are going to write a few functions using OCaml's imperative features.

1. [**20**]

   Write two imperative versions of the fibonacci function that takes a single `int` argument `n` and returns an `int` result, which is the `n`th fibonacci number. The first one should be called `fibonacci` and uses a `while` loop, while the second is called `fibonacci2` and uses a `for` loop. **Do not use recursion or any helper functions in either problem.** The resulting functions should be O(`n`). The functions should store all data in reference cells (except for the argument `n` and loop indices in `fibonacci2`) and update them as necessary.

   Hint: Write these functions basically the way you would write them in C or Python (modulo syntax, of course).

2. [**30**]

   We saw the bubble sort algorithm in the midterm. This algorithm is normally implemented as an imperative algorithm, which we will do here. It works on arrays of values, sorting in-place in ascending order. The `bubble_sort` function has the signature:

   ```
   val bubble_sort : 'a array -> unit
   ```

   You may assume that the type `'a` is orderable *i.e.* that operators like `<`, `=`, and `>` work properly on values of that type.

   Again, do not use recursion or any helper functions in your function. Use `for` loops or `while` loops for looping.

   Imperative bubble sort works as follows. Go through the array, looking at consecutive pairs of elements. If you find one that is out of order (the second element is smaller than the first), swap the two elements in-place (you might want to use a temporary variable for this) and continue. Once you reach the end of the array, the last element of the array should be the largest. Start over and continue until all the elements are in the correct order. This algorithm has a time complexity of O(n^2).

   Here is a reference on bubble sort.

Part B: Tagged Data

In this problem we will extend the tagged data representation of units that we explored in lecture 10.

a. **[10]**

Recall the code we ended up with at the end of the lecture:

```
let meters_per_foot = 0.3048

let get_meters len =
  match len with
    | `Meter m -> m
    | `Foot f -> f *. meters_per_foot

let length_add a b = `Meter (get_meters a +. get_meters b)
```

This code is nice in that it makes it easy to add new unit types without modifying a lot of code. Show that this is right by modifying the code to work with inches (where a length in inches will be represented by the tag `` `Inch `` *e.g.* `` `Inch 2.3 ``). You should only have to add a single line to one of the functions above. *Hint:* There are 12 inches in a foot :-)

b. **[15]**

By analogy with the length abstraction given above, create a mass abstraction using tagged data. Mass units you should handle include grams, kilograms, and slugs (using the variant tags `` `Gram ``, `` `Kilo ``, and `` `Slug ``, respectively). Note that a slug is 14593.903203 grams. Write a function called `get_grams` which returns the gram equivalent of any mass value as a float. Use this to write a function called `mass_add` which will add together any two kinds of masses.

Repeat this process by defining a time abstraction using tagged data. Time units you should handle include seconds, minutes, hours, and days (using the variant tags `` `Second ``, `` `Minute ``, `` `Hour ``, and `Day`, respectively). Write a function called `get_seconds` which returns the seconds equivalent of any time value as a float. You can look up the conversion factors if you don't know them. Use this to write a function called `time_add` which will add together any two kinds of times.

c. **[10]**

Now that we have length, mass, and time abstractions, we can deepen the abstraction hierarchy and write a generic `unit_add` function which will take any two tagged data values and add them if they are compatible or signal an error (using the `failwith` function) if not. For this, assume that every tagged data value is tagged not just with its unit tag but with a unit "class" tag. So a length unit wouldn't be just `` `Meter 1.0 `` but `` `Length (`Meter 1.0) ``, and similarly for other united data values. The unit class variant tags are: `` `Length ``, `` `Mass ``, and `` `Time ``.

Do we get into a combinatorial explosion when adding more unit classes? Why or why not? Write your answer in an OCaml comment.

Part C: Message Passing

1. **[30]**

Give a message-passing implementation for a gram unit which supports the following messages:

a. `get_grams`

   Returns a number (OCaml float) representing the mass in grams.

b. `get_slugs`

   Returns a number (OCaml float) representing the mass in slugs. Note that a slug is roughly equal to 14593.903203 grams.

c. `unit_type`

   Returns `` `Gram ``.

d. `compatible`

   Returns `true` when given anything which the `add` message of the gram message-passing object supports (*i.e.* a gram or a slug); here is an example:

```
(* NOTE: You do not have to define make_slug or make_foot.
   This is an example to show how it should work. *)

let agram = make_gram 1.0
let bslug = make_slug 1.0
let cfoot = make_foot 1.0

agram#compatible agram  (* true *)
agram#compatible bslug  (* true *)
agram#compatible cfoot  (* false *)
```

   Hint: You can assume that `make_slug` and `make_foot` return message-passing objects that accept the `unit_type` message.

e. `add`

   Adds a compatible mass, returning a message-passing mass object. Here's an example:

```
let agram = make_gram 1.0
let bslug = make_slug 1.0  (* similarly, you do not have to define make_slug *)
let cgram = agram#add bslug
cgram#get_grams  (* 14594.9 (grams) *)
```

   An incompatible object passed in as an argument should cause an error to be signalled using the `failwith` function.

   Here is a template for your code:

```
let rec make_gram g =
  let
    ... (* internal definitions *)
  in
    object
      method get_grams = ...
      method get_slugs = ...
      method unit_type = ...
      method compatible other = ...
      method add other = ...
    end
```

   Fill in the parts marked "`...`". This requires less than 20 lines of code.

2. **[60]**

In this problem, we will consider a message-passing implementation of the differentiator from lab 4. In addition to taking the derivative, it will be possible to convert an expression to a string and evaluate an expression. For evaluation, we will provide a value for a variable and compute the expression (or number) that results from substituting in the number for the value (much like evaluation in the substitution model, **except** that we will only be giving a value to one variable at a time, so we must keep track of symbolic expressions involving the variables which have not been given values).

Each expression will be a message passing object which responds to the following messages:

- `value`

  Return the value of an expression as an OCaml `int` (only valid for expressions which represent numbers; anything else should signal an error).

- `show`

  Return a representation of the expression as a string. Operator expressions should be wrapped in parentheses so we don't have to worry about precedence.

- `is_zero`

  Return `true` if the expression is the number 0.

- `is_number`

  Return `true` if the expression is a number.

- `derive`

  Return the derivative of the expression with respect to the given variable. This derivative will be a message-passing object, just like the original expression was.

- `evaluate`

  Evaluate the expression for a given variable and value. Note that this will not necessarily be the most simplified version of that expression, but it must be correct.

Examples:

```
(* g represents 5*x + x*y + 7*y *)
# let g =
  make_sum
    (make_product
      (make_number 5)
      (make_variable "x"))
    (make_sum
      (make_product
        (make_variable "x")
        (make_variable "y"))
      (make_product
        (make_number 7)
        (make_variable "y"))) ;;
val g :
  < derive : string -> 'a; evaluate : string -> int -> 'a; is_number :
    bool; is_zero : bool; show : string; value : int >
  as 'a = <obj>
```

```
# g#show;;
- : string = "((5 * x) + ((x * y) + (7 * y)))"
(* Note that these results are not maximally simplified. *)

# (g#evaluate "x" 2)#show;;
- : string = "(10 + ((2 * y) + (7 * y)))"

# (g#evaluate "y" 3)#show;;
- : string = "((5 * x) + ((x * 3) + 21))"

# ((g#evaluate "x" 2)#evaluate "y" 3)#show;;
- : string = "37"

# (g#derive "x")#show;;
- : string = "(5 + y)"

# (g#derive "y")#show;;
- : string = "(x + 7)"

# (g#derive "z")#show;;
- : string = "0"

# (((g#derive "x")#evaluate "x" 2)#evaluate "y" 3)#show;;
- : string = "8"
```

Starting Code base:

```
(* Define a number as a message-passing object. *)
(* "i" is an int. *)
let rec make_number i =
  object
    method value = i
    method show = string_of_int i
    method is_zero = i = 0
    method is_number = true
    method evaluate _ _ = make_number i  (* must evaluate to an object *)
    method derive _ = make_number 0  (* derivative of a number is 0 *)
  end

(* Define a variable as a message-passing object. *)
(* "v" is a string. *)
let rec make_variable v =
  object
    method value = failwith "variable has no numerical value"
    method show  = v
    method is_zero = false
    method is_number = false
    method evaluate v' n =
      if v = v'
        then make_number n
        else make_variable v
    method derive v' =
      if v = v'
        then make_number 1  (* d/dx(x) = 1 *)
        else make_number 0  (* d/dx(y) = 0 *)
  end

(* Define a sum as a message-passing object. *)
let rec make_sum expr1 expr2 =
  match () with
    | _ when expr1#is_zero -> expr2  (* 0 + expr = expr *)
    | _ when expr2#is_zero -> expr1  (* expr + 0 = expr *)
```

```
    | _ when expr1#is_number && expr2#is_number ->  (* add numbers *)
        make_number (expr1#value + expr2#value)
    | _ ->  (* create a new object representing the sum *)
        object
          method value = failwith "sum expression has no numerical value"
          method show = "(" ^ expr1#show ^ " + " ^ expr2#show ^ ")"
          method is_zero = false
          method is_number = false
          method evaluate v n =
            make_sum (expr1#evaluate v n) (expr2#evaluate v n)
          method derive v =
            make_sum (expr1#derive v) (expr2#derive v)
        end

(* Evaluate a message-passing expression with a number
   substituted for a variable. *)
let evaluate expr v n = expr#evaluate v n

(* Return the string representation of an expression. *)
let show expr = expr#show

(* Return the derivative of an expression. *)
let differentiate expr v = expr#derive v
```

---

a. **[90]** Write a function called `make_product` to create message-passing product expressions (one expression multiplied by another) in a manner analogous to `make_sum`. It should create a single numeric result when possible (*i.e.* when all the argument expressions given are numbers). Multiplication by the number zero should be reduced to the number zero, and multiplication of a term by one should be reduced to the term. You can use the `make_sum` function above as a skeleton to build your solution around, but be careful to notice the differences between sums and products (for one thing, there are more base cases with products). Note that the derivative of a product of two expressions can be computed with the following rule:

```
Deriv(f(x) * g(x)) = Deriv(f(x)) * g(x) + f(x) * Deriv(g(x))
```

where the derivatives are all with respect to `x`.

This function can be written in less than 30 lines.

b. **[15]** Demonstrate that your operations work by evaluating the following expressions in the interactive OCaml interpreter and showing your results (in OCaml comments).

1.
```
(* f = x^3*y + 3*x^2*y^2 + y^2 + 2 *)
let f =
  make_sum
   (make_product
    (make_variable "x")
    (make_product
     (make_variable "x")
     (make_product
      (make_variable "x")
      (make_variable "y"))))
   (make_sum
    (make_product
     (make_number 3)
     (make_product
      (make_variable "x")
      (make_product
       (make_variable "x")
       (make_product
        (make_variable "y")
```

```
          (make_variable "y"))))))
      (make_sum
       (make_product
        (make_variable "y")
        (make_variable "y"))
       (make_number 2)))
```

2. `# let dfdx = differentiate f "x";;`
3. `# show dfdx;;`
4. `# show (evaluate f "x" 3);;`
5. `# show (evaluate (evaluate f "x" 3) "y" 4);;`
6. `# show (evaluate (evaluate dfdx "x" 3) "y" 4);;`

Hint: You probably want to test your functions on some simpler operations first. Here are some examples. You do not have to submit the results of using your functions on these values.

```
let n0  = make_number 0
let n1  = make_number 1
let n2  = make_number 2
let x   = make_variable "x"
let s1  = make_sum n1 n2
let s2  = make_sum n1 n0
let s3  = make_sum x n1
let s4  = make_sum (make_variable "y") (make_number 4)
let p1  = make_product n2 n2
let p2  = make_product x n0
let p3  = make_product x n2
let p4  = make_product x s1
let sl1 = make_sum p3 (make_sum p4 s3)
let p5  = make_product n2 p4
let p6  = make_product x s3
let ap1 = make_sum p3 p4
let pa1 = make_product s3 s4
let pl1 = make_product s2 (make_product s4 sl1)
```
