CS 4: Fundamentals Of Computer Programming, Winter 2017

Assignment 6: Environmentally-Friendly

Due: *Friday, March 3, 02:00:00*

Coverage

This assignment covers the material up to lecture 14, as well as the material in recitation lectures 4 and 5.

What to hand in

All of your code should be saved to a file named `lab6.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: `lab6.mli`)
2. a test script: `tests_lab6.ml`)
3. a `Makefile`.)

You should download all of these files into the directory in which you are writing your `lab6.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, type this:

```
$ make test
```

This will compile the `tests_lab6.ml` file (which contains the unit tests) and output an executable program called `tests_lab6`. Then it will run that program and report all test failures.

If you want to compile the code and immediately run the 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
```

Environment descriptions

There is a problem in part A that requires you to write a textual description of the environments generated by evaluating some code using the environment model described in lecture 13. The approach taken in previous years required students to actually use a drawing program and draw diagrams like the ones in lecture 13, but this was (to put it mildly) not very popular, so we now use a different approach. We'll describe this process here.

Consider the following OCaml code:

```
let square x = x * x in square 100
```

We would like to evaluate this using the environment model, and in the process describe the environments created during its evaluation. This was done in lecture 13 using diagrams, but note that you can do the same thing using the following text description:

```
(*

  -- Desugar expression to:

  let square = fun x -> x * x in square 100

  -- Start with initial environment:

  FRAME 0 (initial environment)
    parent: none
    bindings:
      * : [primitive function *]

  -- Evaluate the let expression:
  --   Evaluate the fun expression:

  FUNCTION 0 (fun x -> x * x)
    env: FRAME 0    (* environment function was defined in
                       AKA enclosing environment *)
    param: x
    body: x * x

  --   Create a new frame with the name "square" bound to the function object:

  FRAME 1 (let square = FUNCTION 0 in ...)
    parent: FRAME 0
    bindings:
      square : FUNCTION 0

  --   Evaluate the expression "square 100" in the context of FRAME 1:
  --     100 evaluates to itself
  --     Look up "square" --> FUNCTION 0
  --     Apply FUNCTION 0 to 100:
  --       Create a new frame, binding the formal parameter of the "square"
  --         function:

  FRAME 2  (FUNCTION 0 applied to 100)
    parent: FRAME 0
    bindings:
      x : 100
  -- Note that FRAME 2's parent is FRAME 0 because FUNCTION 0's enclosing
  -- environment is FRAME 0.

  --       Evaluate "x * x" in the context of FRAME 2:
  --         Look up x: 100  (from FRAME 2)
  --         Look up *: [primitive function *]  (from FRAME 0 via FRAME 2)
  --         Apply * to 100, 100 --> 10000
  --         RESULT: 10000

*)
```

The environment description then would consist of all frames and functions created during this evaluation:

```
  (*

    FRAME 0 (initial environment)
      parent: none
      bindings:
        * : [primitive function *]

    FUNCTION 0 (fun x -> x * x)
      env: FRAME 0
      param: x
      body: x * x

    FRAME 1 (let square = FUNCTION 0 in ...)
      parent: FRAME 0
      bindings:
        square : FUNCTION 0

    FRAME 2  (FUNCTION 0 applied to 100)
      parent: FRAME 0
      bindings:
        x : 100

  *)
```

This is the kind of description we want you to produce. You don't have to describe the evaluation process step-by-step, but you need to be able to do it in your head (or on paper), or you won't get the right frame/function contents. Pay particular attention to parents of frames; it's extremely easy to get those wrong. Remember: applying a function to its arguments creates a new frame whose parent frame is the same as the `env` (enclosing environment) of the function being applied.

Note that primitive functions don't create frames when applied. Also, only put the primitive functions used in the evaluation into the initial environment (even though in reality there would be dozens more).

Working through problems like this will very rapidly clarify your understanding of the environment model.

More on objects

Part B requires that you use a couple of features of OCaml's object system that haven't been needed so far: self references and private methods. They are described here. Self-references will also be described in class, as they interact with the environment model (technically, they enable what is called *open recursion*).

Self references

One common thing to want to do in any object-oriented language is to be able to call methods in an object from other methods. Typically there is a dedicated keyword for "the current object" (usually `self` or `this`), but OCaml has a more flexible solution. If you need to identify the current object, you just put its name in parentheses after the keyword `object`. So instead of writing:

```
object
  (* method definitions *)
end
```

you write:

```
object (self)
  (* method definitions *)
end
```

and then you can call methods in the object from within other methods in the same object using the syntax `self#foo` for a method `foo`. Note that you don't have to use the word `self`; you can write:

```
object (this)
  (* method definitions *)
end
```

if you prefer, or even:

```
object (fnordly)
  (* method definitions *)
end
```

(but if you do the latter, you will annoy your grader!). We recommend you stick with `self`, because it's what everyone else does.

Private methods

It's quite common when defining objects to want some methods to be restricted to the object implementation itself. Self-references allow the calling of one method in an object from another, but to have private methods requires that the method definition be marked `private`. So instead of this:

```
method foo = ...
```

you would write:

```
method private foo = ...
```

When this is done, the method `foo` can only be called by other methods in the same class (unless inheritance is used, in which case it can also be called by subclasses, but we haven't covered inheritance).

Part A: The Environment Model

In this section, we will explore the environment model described in lecture 13.

1. [**60**] Adapted from SICP, exercise 3.9.

   Consider this code:

   ```
   let factorial n =
     let rec iter m r =
       if m = 0
         then r
         else iter (m - 1) (r * m)
   ```

```
    in iter n 1
  in
    factorial 3
```

Write a textual description of the environment (all frames and functions) created while evaluating this code according to the environment model. See the "OCaml notes" section for guidelines on how this is to be done. **Note:** This is a text-based description only! Write your answer as an OCaml comment.

To give you a head start, here is the initial environment:

```
(*

  FRAME 0 (initial environment)
    parent: none
    bindings:
      - : [primitive function -]
      * : [primitive function *]

*)
```

Note that you do *not* have to write out a detailed evaluation of the code (though you can if you want), just the frames and functions generated in its evaluation. Please indicate which code created each frame/function in parentheses beside the FRAME/FUNCTION name and number. You can use ellipses (`...`) if the code is too long to comfortably fit on the line. You can also refer to previously-defined functions as *e.g.* `FUNCTION 0` instead of copying the code; see the "OCaml notes" section for examples.

Note also that you don't have to desugar functions of multiple arguments into their curried version (`fun x y -> ...` to `fun x -> fun y -> ...`). Similarly, when applying a multi-argument function to its arguments you can create a single frame with bindings for all function arguments.

When writing out function descriptions, include the enclosing environment (`env`), the formal parameter(s) (`params`) and the body of the function (`body`). If the body is too long to fit on the line, use ellipses (`...`) to shorten it.

Other notes:

1. There are exactly two functions and eight frames in the final solution (including FRAME 0 shown above).
2. Make sure to desugar function definitions to their corresponding `fun` forms. Other than that, don't desugar `let` expressions to `fun` expressions. Especially don't do this with `let rec` expressions, because it isn't valid!
3. The final result should be 6 :-) You don't have to write this out, though.

2. [**20**]

Ben Bitfiddle doesn't like environment model rule 4, which says how to evaluate a `let rec` expression. "It's too complicated!" he says. "I can achieve the exact same thing using `let` and `ref` cells." He points out a typical piece of OCaml code:

```
let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1)
```

Then he starts to redo it his way to eliminate the `let rec`:

```
let factorial =
  let f = ref (fun n -> 0) in
```

Complete Ben's code to make a working `factorial` function without using `let rec`. You only have to add a few more lines. *Hint:* Consider the original factorial function where the recursive call is replaced by something else. *Hint 2:* This is actually similar to the way we described how `let rec` works in the environment model, with the ref cell initially holding the dummy value. The value `fun n -> 0` is there so that the code type checks; that function will never be applied.

Part B: Message Passing and mutation

The *mean*, *variance*, and *standard deviation* are three useful statistical values often computed over a set of numeric data. If you don't know what these terms refer to, look them up in Wikipedia. We will be using the "population variance" and "population standard deviation" in what follows; in other words, divide by `n` instead of by `n - 1` in the denominators of the variance and standard deviation, where `n` is the number of data values.

Often, when collecting data, we receive it one element at a time instead of all at once. It might, then, be useful to be able to determine the mean, variance, and standard deviation "on-the-fly" as we add new values to the object. (Many scientific calculators let you do this.)

Your job in this section is to create message-passing *statistics objects* to provide this functionality. You will explore OCaml's object-oriented programming features by writing two variants of the statistics object. This will also give you experience with using object-oriented programming in conjunction with mutation, which is a very common programming scenario. (In fact, it could be argued that one of the main purposes of object-oriented programming is to make mutation more manageable.)

Your statistics objects should adhere to the following interface:

- The constructor function should be called `make_stat_N`, where `N` will be either 1 or 2, and should take a single argument of type `unit`. The constructor should create a new statistics object with no data, but which is capable of storing data and computing statistical summaries of input data.

- You should provide a method called `append` to insert an additional data value into the object. This method takes a single argument (the new data value, which must be a `float`). It updates the internal state variables (see below) to their new values.

- You should provide methods to obtain the current mean, variance, and standard deviation of the numbers previously appended to the object: `mean`, `variance`, and `stdev` respectively. Calling these methods on a statistics object to which no data has been appended should result in a `Stat_error` exception being raised with an error message (you will have to define `Stat_error` as a new exception type yourself). These methods take no arguments.

- You should have a `clear` method which resets all state variables to their original values. This method takes no arguments.

Use the following equations to compute the mean, variance, and standard deviation given the sum, sum of squares, and number of items:

```
mean = sum / n  (* n is the number of items *)
variance = ((sum of squares) - (sum * sum / n)) / n
standard deviation = square root(variance)
```

Hint: A key aspect of designing a message-passing object is knowing where to put the state variable(s). In this case, use `let` expressions outside of the `object` expression to store the state variables. The environment model will show you why this works. Putting the `let` inside the `object` will **not** work.

You should have the following state variables:

1. `sum`: the sum of all the appended values
2. `sumsq`: the sum of the squares of all the appended values
3. `n`: the number of values that have been appended

All state variables are initially zero. `sum` and `sumsq` are references to `float`; `n` is a reference to an `int`. All of these are reference cells because they need to be updated as new data comes in.

Note that there will *not* be a state variable to hold all the data values that have been added to the list. This would be extremely costly in terms of space usage, and wouldn't help us to compute any of the statistics we want. Therefore, when a new value is "appended" to the object, all that happens is that the state variables are updated as needed; the actual data value itself is not stored.

Note: Do not write any helper functions for your statistics objects. This may seem counterproductive, but there is a larger point we are trying to make. This means that there will be some unavoidably duplicated code in some of the object implementations.

Here are the two different statistics objects you need to write:

1. **[30]** The first statistics object will be created by a call to the constructor function `make_stat_1`. This function will return an object which supports all the methods described above.

   For this problem, we don't want you to use a `self` reference in the object or private methods. This will mean that you will have to duplicate some code in some of the methods.

   Here are some example interactions with one of these statistics objects.

```
# let mystat = make_stat_1 ();;
val mystat :
  < append : float -> unit; clear : unit; mean : float; stdev : float;
    variance : float > =
  <obj>
# mystat#mean;;
Exception: Stat_error "need at least one value for mean".
# mystat#variance;;
Exception: Stat_error "need at least one value for variance".
# mystat#stdev;;
Exception: Stat_error "need at least one value for stdev".
# mystat#append 1.0;;
- : unit = ()
# mystat#append 2.0;;
- : unit = ()
# mystat#append 3.0;;
- : unit = ()
# mystat#mean;;
- : float = 2.
# mystat#append 4.0;;
- : unit = ()
# mystat#variance;;
- : float = 1.25
# mystat#stdev;;
- : float = 1.1180339887498949
# mystat#clear;;
- : unit = ()
# mystat#append 1.0;;
- : unit = ()
# mystat#mean;;
- : float = 1.
# mystat#variance;;
- : float = 0.
```

```
# mystat#stdev;;
- : float = 0.
```

2. **[15]** The second statistics object will be created by a call to the constructor function `make_stat_2`. We will take this opportunity to improve the code in `make_stat_1`, though the examples given above will still work and will give the same results.

   Most of this code will be identical to that in `make_stat_1`. We want you to make the following changes:

   1. Aside from error handling and a single call to `sqrt`, the code in the `variance` and `stdev` methods is almost identical. Factor out the similar differences into a method called `_variance` that computes the variance assuming that there has been at least one number appended to the stats object (in other words, `_variance` will not do any error checking). Add a `self` reference to the `object` form (see the OCaml notes section above for details on this) and call the `_variance` method from both the `stdev` and `variance` methods to compute the variance where needed.

   2. Since the `_variance` method is not part of the object's interface, make it a `private` method.

Part C: Modules and functors

In this section, you will work with OCaml modules and functors by implementing a data structure called a priority queue. The specific implementation you will create is called a leftist heap. So let's talk about these things now.

Priority queues

NOTE: There is a priority queue implementation given in the OCaml manual as part of the module documentation. Please don't consult that for this assignment (the implementations are different anyway).

A priority queue is a data structure that is ordered so that it provides fast access to its minimum element, as defined by some ordering relationship. Conceptually, you can think of that element as "the next thing to be processed". It also has to have an operation which deletes the minimum element (which would be useful after processing that element) and returns another priority queue, as well as several other operations, which are summarized here as an OCaml module type:

```
module type PRIORITY_QUEUE =
  sig
    exception Empty

    type elem      (* Abstract type of elements of queue. *)
    type t         (* Abstract type of queue. *)

    val empty      : t                 (* The empty queue.         *)
    val is_empty   : t -> bool         (* Check if queue is empty. *)
    val insert     : t -> elem -> t    (* Insert item into queue.  *)
    val find_min   : t -> elem         (* Return minimum element.  *)
    val delete_min : t -> t            (* Delete minimum element.  *)
    val from_list  : elem list -> t    (* Convert list to queue.   *)
  end
```

Of these operations, note that the operation `from_list` is not fundamental to the definition of a priority queue, but it's extremely useful.

Also note that we are being loose about the words "insert" and "delete". These operations are purely functional, so that when we say we "insert" an element into a queue, we really mean that we create a new queue which has the extra element in it; the original queue is unchanged. Similarly, when we "delete" an element from a queue, we really mean that we create a new queue without the offending element; the original queue is once again unchanged.

Leftist heaps

Now that we know how our data structure is supposed to behave, the next question is: how do we implement it? Naturally, there are lots of choices, each of which will have different trade-offs. In this section you're going to implement priority queues as *leftist heaps*. This is a data structure that has the following attributes:

- It can either be a leaf (representing an empty heap) or a node which contains:
    - a stored element
    - a non-negative integer value called its "rank"
    - a left and right subheap, each of which is also a leftist heap

  Thus, a leftist heap is a binary tree with additional rank information stored in the nodes. The tree will in general **not** be balanced; it will usually have more elements in the left subtree than in the right. We say that the tree "skews to the left".

- The element stored at the top of the tree is smaller (more precisely, is no larger) than any of the elements stored beneath it in either subtree.

- As mentioned, each node has an integer value associated with it, which is called its *rank*. The rank of a node is equal to the length of its *right spine*, which is the number of nodes in the rightmost path from the node in question to an empty (leaf) node (this will also turn out to be the shortest path to a leaf node). Thus, a leaf node has rank 0, and a heap with a left and/or right subheap which is a leaf has rank 1. However, ranks are only stored in nodes, not in leaves. Having the rank stored in the nodes makes many operations much faster.

- The rank of any left child is required to be at least as large as the rank of its right sibling. This is why it's called a "leftist" heap.

The interesting feature of leftist heaps is that they support very efficient merging of two heaps to form a combined heap. (Of course, since we're using purely functional code, you don't alter the original heaps.) Leftist heaps can be merged in O(log N) time, where N is the total number of elements in the resulting heap. Furthermore, once the merge operation (which, you'll note, is not a required operation for priority queues) is implemented, you can define most of the rest of the operations very easily in terms of it. Specifically, you can define the `insert` and `delete_min` operations in terms of merging, and the other operations are then trivial to define, except for the list-to-heap conversion routine. That can be done easily in O(N * log(N)) time, and with more difficulty in O(N) time. You're not required to find the most efficient solution, but it's a good exercise.

Merging leftist heaps

OK, so now we need to figure out how to merge two leftist heaps to create a new leftist heap. The basic algorithm is quite simple:

- If either heap is empty, return the other heap.

- If the first heap's minimum element is smaller than the second heap's minimum element, make a new heap (*) from the first heap's minimum element, the first heap's left subheap, and the result of merging the first heap's right subheap with the second heap.

- Otherwise, make a new heap (*) from the second heap's minimum element, the second heap's left subheap, and the result of merging the first heap with the second heap's right subheap.

(*) Here is how to make a new heap from a minimum element and two heaps: the resulting heap must have:

- the given minimum element
- a rank which is the smaller of the ranks of the original heaps, plus 1
- a left subheap which is the original heap with the larger rank
- a right subheap which is the original heap with the smaller rank

This algorithm will preserve the leftist heap property in the merged heap.

Here are the problems:

1. [**60**]

   Write a module `PriorityQueue` which implements the `PRIORITY_QUEUE` module type. Assume that the `elem` type will be `int`. Here is a template for your code:

   ```
   module PriorityQueue : (PRIORITY_QUEUE with type elem = int) =
     struct
       (* Your code goes here. *)
     end
   ```

   You will need to fill in the definitions for all the module functions, as well as the types and exceptions. The exception `Empty` is trivial; it's the same as it is in the module type. The queue type `t` should be written to conform with the description above.

   You can also write additional unexported helper functions in the module, if that will make your life easier. Make sure you raise the `Empty` exception when trying to do invalid things *e.g.* finding the minimum value in an empty queue.

   Note that since the type `elem` is abstract, if you want to use a real type as the elements of your priority queue (and it would be pretty useless otherwise), you have to specify which type you want the `PRIORITY_QUEUE`'s `elem` value to represent. This is kind of clunky syntax. Also, you might ask why you can't just make it parametric, like a type `'a list`. You actually can in some cases, but here we need a type with an ordering relation, and there is no way to guarantee that any arbitrary type will (a) be orderable at all – what if it's a function type? or (b) will be orderable using the same function (*e.g.* the built-in `compare` function). So it's better to write the code as it's written above. Also, this will make it easy to transform into a functor (see below).

   Here's an interesting point to ponder: why don't we also have to specify what the abstract priority queue `t` type represents in the code above?

   Use the built-in (overloaded) comparison function `compare` and/or the comparison operators (`<`, `>` *etc.*) to do whatever comparisons may need to be done. This works because OCaml's comparison operators work on any type. (We'll see a more elegant way to do this below.)

   Use your priority queue implementation to write a *heap sort* function called `heap_sort`. This will take a list of `int`s as its argument and will

   - first convert the list to a priority queue (implemented as a leftist heap) using the `from_list` function of the module
   - successively remove the smallest element from the heap and "cons" it onto (*i.e.* place it onto the front of) a list until there are no more items in the heap
   - reverse the list to get a sorted list in ascending order

Only use your module's exported functions (those in the module type) in your solution.

2. [**30**]

As written, the code is dependent on the built-in comparison functions. To make this more generic, let's define some types and module types:

```
(* Type for ordered comparisons. *)
type comparison = LT | EQ | GT

(* Signature for ordered objects. *)
module type ORDERED =
  sig
    type t
    val cmp: t -> t -> comparison
  end

(* Signature for priority queues. *)
module type PRIORITY_QUEUE =
  sig
    exception Empty

    type elem
    type t

    val empty      : t
    val is_empty   : t -> bool
    val insert     : t -> elem -> t
    val find_min   : t -> elem
    val delete_min : t -> t
    val from_list  : elem list -> t
  end
```

What you have to do now is generalize your priority queue implementation into a functor that takes a module implementing the ORDERED module type as its argument, and produces a priority queue (implemented using a leftist heap) which is specialized for that particular type of data. For instance, you can define a module of ordered strings like this:

```
module OrderedString =
  struct
    type t = string
    let cmp x y =
      if x = y then EQ else if x < y then LT else GT
  end
```

and then define your string priority queue like this:

```
module StringPQ = MakePriorityQueue(OrderedString)
```

Here, `MakePriorityQueue` is in fact a functor. It takes an input module (in this case, `OrderedString`), and returns an output module (in this case `StringPQ`) which is a priority queue specialized to work only on strings.

Once you've done this, redefine your heap sort function using a StringPQ as the heap. Call the new function `heap_sort_2`. Note that this heap sort will only work on strings.

To get you started, here is a skeleton of the code you should use for the functor definition:

```
module MakePriorityQueue (Elt : ORDERED)
  : (PRIORITY_QUEUE with type elem = Elt.t) =
  struct
```

```
      (* Your code goes here... *)
    end
```

Note that again you have to specify what the type `elem` in the `PRIORITY_QUEUE` represents. Here, it better be the same type as the type `t` in the `ORDERED` argument (which we have called `Elt`).

This code will be nearly identical to the code above. All you are really doing is repackaging that code into a functor.