CS 4: Fundamentals Of Computer Programming, Winter 2017

Assignment 7: Final preparation

Due: *Friday, March 10, 02:00:00*

Coverage

This assignment covers the material up to lecture 16 plus all recitation lectures.

Notes

There will be no opportunity for rework on this lab assignment. As a result, it will *not* be graded on a minimum-score scale. More importantly, though, late submission on this assignment may make it difficult or impossible for the TAs to grade it in a timely manner. Please make every effort to complete the assignment on time.

What to hand in

All of your code should be saved to a file named `lab7.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: `lab7.mli`)

2. a test script: `tests_lab7.ml`)

3. a `Makefile`.)

4. a template file called lab7.ml.

You should download all of these files into the directory in which you are writing your code. You should not change anything but the `lab7.ml` file; please do not submit the other files as part of your assignment submission to csman! For the `lab7.ml` file, please write your code where you see the `(* TODO *)` comments in the code; also remove the `(* TODO *)` comments. **Failure to remove the `(* TODO *)` comments will cause you to lose marks!**

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_lab7.ml` file (which contains the unit tests) and output an executable program called `tests_lab7`. 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
```

OCaml arrays

OCaml supports arrays as a built-in datatype. Arrays are always mutable and always contain exactly one type of data, so the (polymorphic) array type is referred to as `'a array`. More specifically, we can have `int array`, `float array`, `string array`, or even arrays of arrays (*e.g.* `int array array` would be a two-dimensional array of `int`s).

Arrays can be constructed using the array literal syntax, which is the same as the syntax for lists except that instead of square brackets `[` and `]` you use "array brackets", which are `[|` and `|]`. Here are some OCaml arrays and some array operations:

```
# [| 1; 2; 3; 4; 5 |] ;;
- : int array = [|1; 2; 3; 4; 5|]
# [| 1.2 +. 2.3; 4.0 -. 5.6; 7.8 +. 100.2 |] ;;
- : float array = [|3.5; -1.59999999999999964; 108.|]
# let arr = [| 1.2; 2.3; 3.4; 4.5; 5.6 |] ;;
val arr : float array = [|1.2; 2.3; 3.4; 4.5; 5.6|]
# Array.length arr;;
- : int = 5
# arr.(0);;
- : float = 1.2
# arr.(4);;
- : float = 5.6
# arr.(5);;
Exception: Invalid_argument "index out of bounds".
# arr.(-1);;
Exception: Invalid_argument "index out of bounds".
# let arr2 = [| [| 1; 2; 3 |]; [| 4; 5; 6 |]; [| 7; 8; 9 |] |] ;;
val arr2 : int array array = [|[|1; 2; 3|]; [|4; 5; 6|]; [|7; 8; 9|]|]
# arr2.(0).(0);;
- : int = 1
# arr2.(0).(2);;
- : int = 3
# arr2.(2).(0);;
- : int = 7
# arr2.(2).(2);;
- : int = 9
```

Indexing arrays uses the notation `arr.(i)` with `i` an `int`; don't use square bracket notation here! For two-dimensional arrays, use `arr.(i).(j)`, *etc.*. Use `Array.length` to get the length of an array.

Mutating elements in arrays uses this syntax:

```
# arr.(0);;
- : float = 1.2
# arr.(0) <- 4.5;;
- : unit = ()
# arr.(0);;
- : float = 4.5
# arr2.(0).(0);;
- : int = 1
# arr2.(0).(0) <- 42;;
- : unit = ()
# arr2.(0).(0);;
- : int = 42
```

The OCaml standard library has an `Array` module with many useful functions on arrays.

Loops and arrays

Note that `for` loops in OCaml are very useful for iterating through arrays. You can also use recursive functions, but `for` loops are more concise.

For loops have the form:

```
for (* variable *) = (* start *) to (* end *) do
  (* statements *)
done
```

Note that the variable takes on all values from `<start>` to `<end>`, *including* the endpoint (this is different from most programming languages). You can have multiple statements inside the `for` loop, separated by semicolons.

When you use `for` loops, you may want to break out of a loop prematurely. OCaml does not have a `break` statement, but you can get the same effect using exceptions. The `Exit` exception is normally used for this; it takes no arguments. Breaking out of a loop would then be implemented like this:

```
try
  for i = 1 to 10 do
    if (* some condition *)
      then raise Exit (* break out of loop *)
      else (* other code *)
  done
with Exit -> (* other code *)
```

Modules and functors

We saw OCaml modules and functors last assignment. We will continue to use them this assignment, and also on the final exam.

The most common use of modules is implicit: whenever you write code in a `.ml` file, that file becomes a module (much like the case with Python). For each `.ml` file, you typically have a `.mli` file giving the module's interface (type signatures for all exported values and functions, `type` declarations for all types, `exception` declarations for all new exceptions). As you know, not all functions defined in a `.ml` file need to be exported; functions that are only used internally won't have their signature specified in the `.mli` file. Similarly, you do not have to specify the implementation of a type in the `.mli` file, and often it is better to leave it out (which makes the type *abstract*, meaning that only the functions in the `.ml` file have access to its internal structure).

As we saw in assignment 6, you can also declare new modules inside of a `.ml` file. These are effectively "modules within modules", and they can be exported (specified in the `.mli` file) or not. The main reason for doing this is when we generate the modules using *functors*. A functor (as we discussed in recitation 5) is a kind of "function on modules" (not a real OCaml function, but conceptually like a function) by which we mean a way to take an existing module and create a new module which uses the existing one. Functor arguments are specified using a `module type`, which is a specification of a particular module (we'll see examples below). A very common use case for functors is to create a data structure from another data structure. For instance, the input module may represent the concrete type of a component of the data structure plus some functions on it (for instance, a function to compare values of that type for orderable types). The functor will take that input module and use it to create a new module representing a data structure containing that type.

All this may seem a bit abstract, so this assignment will show you several examples of functors. Fortunately, most problems with functors have to do with getting the functor syntax right, and we've done all that for you; you just have to fill in the contents. As an example, we will use OCaml sets in this assignment. Sets are defined in OCaml in the `Set` module, which defines a functor called `Set.Make`. This takes as its module argument a module of module type `Ordered_type` which has this signature:

```
module type Ordered_type =
  sig
    type t
    val compare : t -> t -> int
  end
```

This signature (`sig` form in OCaml) says that an ordered type is a type `t` which also has a comparison function called `compare` which takes two values of type `t` and returns an `int`: 0 if the two are "equal", -1 if the first is smaller than the second, and 1 if the first is larger than the second. To create a module that matches this signature, we use the `struct` form (no relation to `struct`s in C/C++!). Here's an example:

```
module OrderedString : Ordered_type =
  struct
    type t = string
    let compare s1 s2 = Pervasives.compare s1 s2
    (* or just: let compare = Pervasives.compare *)
  end
```

This creates an `OrderedString` module which contains the `string` type and a function for string comparison. The `compare` function just dispatches to the OCaml built-in `compare` function, which is more specifically called `Pervasives.compare`; the `Pervasives` module is where all the built-in functions live.

Given this, we can create a set of strings using the `Set.Make` functor as follows:

```
module StringSet = Set.Make(OrderedString)
```

and now we can make sets of strings! The documentation on the kind of sets made by this functor is available here. Here are some examples:

```
# let empty = StringSet.empty;;
val empty : StringSet.t = <abstr>
# let s1 = StringSet.add "foo" empty;;
val s1 : StringSet.t = <abstr>
# let s2 = StringSet.add "bar" s1;;
val s2 : StringSet.t = <abstr>
# StringSet.mem "foo" s2;;
- : bool = true
# StringSet.mem "bar" s2;;
- : bool = true
# StringSet.mem "baz" s2;;
- : bool = false
```

Notice that the set type `StringSet.t` is abstract, which means that once the set has been constructed you can only access it using functions defined in the module. Creating a set of `int`s would be very similar: you would define an `OrderedInt` module and pass it to the `Set.Make` functor to generate an `IntSet` module.

Note that these sets are *functional*. You cannot change the contents of an existing set; you have to create a new one with more or fewer elements. Because of the way that sets are implemented, this can be done quite efficiently.

Also, note that modules are *not* like objects! In particular, a module doesn't store any data. Instead, functions in modules can return values of particular types (like sets) which do store data. Since these types are usually abstract, you can't do anything with that data except pass it to other functions in the module, some of which generally are able to extract components of the data. Thus, modules can do the same kinds of things that objects do, but they do it in a very different way.

Using functors is kind of like riding a bike; at first they seem intimidatingly abstract and weird, but eventually you come to realize that they are just a fairly simple mechanism for splitting up the definition of a module into various reusable parts. Functors are actually one of the great things about OCaml, and it's a pity other programming languages (other than close relatives of OCaml) don't have anything similar.

Useful library functions

There are a number of OCaml library functions that will be useful to you when writing the code for this assignment. Some of these haven't been presented before in this class. Note that you can always browse the OCaml manual and specifically the standard library documentation for more functions to use.

- `List.iter` applies a function to each element of a list, returning a unit value.
- `List.filter` selects values from a list for which a function argument returns `true`.
- `List.fold_left` is like the `accumulate` functions we have seen in the assignments. You don't have to use this, but if used correctly it can make your code shorter.
- `Array.length` returns the length of an array, as an `int`.
- `Array.for_all` checks if every element of an array satisfies a particular condition (represented by a predicate *i.e.* a function that returns a `bool` value).
- `Printf.sprintf` is like `Printf.printf` except that instead of printing a formatted string to the terminal, it returns the string that would have been printed. For instance:

```
# Printf.sprintf "an integer: %d, and a string: %s" 42 "foobar" ;;
- : string = "an integer: 42, and a string: foobar"
```

Miniproject: The game of Lights Out

In this section, you'll implement a simple solitaire game called "Lights Out" (also known as "Lights Off"). The structure of this code is very similar to the way the final exam will be structured; it will give you practice using modules and functors in a functional style to solve a larger-scale problem. Note that this problem involves only writing code to allow a person to *play* Lights Out using the computer; the problem does *not* ask you to write a computer program to *solve* the puzzle! That's an interesting problem, but not what we're looking for here.

For reference, go to this site to familiarize yourself with the game of Lights Out. It's fun! :-) You can also find Flash-based Lights Out games on the internet, and it's probably a good idea to play a couple of games on one of these to familiarize yourself with how the game works.

We will break this problem down into three different kinds of components, all represented as modules or functors.

The first component will represent the Lights Out board, which conceptually is a two-dimensional grid of lights, each of which can be "on" or "off". You will implement two different modules representing boards, each of which will implement the same module type. One module will represent the board as 2-d arrays of boolean values. The other module will represent the board as sets of board locations.

The second component will represent the Lights Out game itself. It will represent the game as a board (one of the boards you just defined), and will provide functions for making moves on the board and initializing the board from a 2-d array of on/off values. This component will be implemented as a functor which takes as its module argument a board module like one of the ones you defined for the first component.

The last component will represent code that allows you to play the game interactively. This is implemented as a functor too, but we are giving you this code, so there is nothing to implement here. Running your code using this module is a good way to test the code (in addition to using the test suite provided). It's also fun!

Rules of the game

Lights Out is a solitaire puzzle game played on a 5x5 grid of lights. At the beginning of the game, some of the lights are set to be "on" while others are "off". The objective is to turn all the lights off. A move of the game consists of choosing a light to toggle. Once this is done, the light goes from on to off (or from off to on), and then toggles all of the orthogonally adjacent lights as well. Note that when we say "orthogonally adjacent" we mean a position which is within one horizontal or vertical location (but not both) of a particular location, and still inside the board. There is no wrap-around, so location (2, 0) is not orthogonally adjacent to location (2, 4).

Play continues until all lights are off. If you get to that point, you have solved the puzzle for that initial configuration of the board.

Here is a sample game. We represent lights that are ON by "`0`", off by "`.`".

```
    0 1 2 3 4
   -----------
0 | O O O . . |
1 | . . . O . |
2 | O O . O O |
3 | . . . O . |
4 | O O O . . |
   -----------

Enter move (row col): 0 1

    0 1 2 3 4
   -----------
0 | . . . . . |
1 | . O . O . |
2 | O O . O O |
3 | . . . O . |
4 | O O O . . |
   -----------

Enter move (row col): 2 3

    0 1 2 3 4
   -----------
0 | . . . . . |
1 | . O . . . |
2 | O O O . . |
3 | . . . . . |
4 | O O O . . |
   -----------

Enter move (row col): 2 1
```

```
    0 1 2 3 4
   -----------
0 | . . . . . |
1 | . . . . . |
2 | . . . . . |
3 | . O . . . |
4 | O O O . . |
   -----------

Enter move (row col): 4 1

    0 1 2 3 4
   -----------
0 | . . . . . |
1 | . . . . . |
2 | . . . . . |
3 | . . . . . |
4 | . . . . . |
   -----------

You win!
```

Note that this is not the only way you could solve this particular board, but it's probably the fastest.

Part A: The board modules

For this problem you will be implementing two different modules, each of which represents a Lights Out board. Both of these modules will implement the following module type:

```
type loc = int * int

module type BOARD =
  sig
    type t
    exception Off_board

    val make : loc list -> t
    val get : t -> loc -> bool
    val flip : t -> loc -> t
    val is_solved : t -> bool
  end
```

The type alias `loc` represents a (row, column) location on the Lights Out board; we will always be using boards of size 5x5. (However, don't use the "magic number" 5 in your code; use `size` instead, which is defined to be 5 in the template file.)

Here is what the contents of the `BOARD` module type represent:

- The type `t` is the board type (which will be different for each of the two modules you implement).
- `Off_board` is an exception raised when accessing locations which are off the board.
- The `make` function takes a list of "on" locations and returns a board. All locations not in the list are assumed to be "off". If any of the locations in the list are invalid (off the board), it raises the `Off_board` exception.
- The `get` function takes a board and a location and returns `true` if that location is "on", otherwise it returns `false` (unless the location is off the board, in which case it raises the `Off_board` exception).

- The `flip` function takes a board and a location and flips the location's light from "on" to "off" or vice versa. Again, if the location is off the board, it raises the `Off_board` exception.

- The `is_solved` function takes a board and returns `true` if all the lights are "off"; otherwise it returns `false`.

NOTE: The `flip` function **only** flips the particular location given; it does **not** flip the orthogonally adjacent locations. That will be done in the game modules we define below.

1. **[60]** Define a module called `ArrayBoard` which will implement the `BOARD` module type with type `t` as a two-dimensional array of booleans. The specific type is as follows:

   ```
   type t = bool array array
   ```

   Each boolean value represents a light at a location; if it is `true`, the light is on, and if it is `false`, the light is off. The location is represented by the array indices of the light. For instance, if a board is called `arr`, then `arr.(0).(0)` is the light at location (row 0, column 0). The row index is the first index, so `arr.(1).(2)` is the light at row 1, column 2.

   Here are some notes on the functions in this module:

   - Don't forget to download the template file and edit it. A lot of code is supplied pre-written in that file.

   - None of these functions need to be very long. If your solution is longer than 25 lines for any of them, you should rethink it.

   - You can define these functions in any order you want. If you want a function to call another function in the same module, though, it has to be defined after the function it calls.

   - You can define extra "helper" functions in the module if you like. However, don't alter the `BOARD` module type.

   - `get`: As mentioned above, if the location is off the board, raise the `Off_board` exception. You might find the `valid_loc` function supplied in the template file to be useful.

   - `flip`: This function (perhaps counterintuitively) doesn't just alter the 2-d array of booleans passed in as its argument. Instead, it must make a *copy* of the 2-d array and alter that (and we will test this!). Put differently, the input array should *not* be altered in any way. The reason for this is that we are implementing a purely functional solution. You may find the `Array.make_matrix` function and `for` loops to be useful for this function.

   - `make`: Again, the `Array.make_matrix` function is your friend here.

   - `is_solved`: OCaml doesn't have a `break` statement for breaking out of loops. See above for a way to achieve the same result using exceptions.

2. **[60]** Define a module called `SetBoard` that implements the `BOARD` module type, but uses sets of locations instead of arrays of booleans to represent the board. Specifically, a board is a set of all "on" locations. ("Off" locations are not explicitly represented except by virtue of not being in the set of "on" locations.) Therefore, a solved board is an empty set. We have defined the `LocSet` module for you in the template file; its definition is as follows:

   ```
   module LocM : Set.OrderedType with type t = loc =
     struct
       type t = loc
       let compare = Pervasives.compare
     end
   ```

```
module LocSet = Set.Make(LocM)
```

Other than that, you have to implement the same four functions as you did for the `ArrayBoard` module. The idea is that a `SetBoard` could be used as a "drop-in" replacement for an `ArrayBoard` and everything will still work.

Here are some notes on this module and its functions:

- The code for this module's functions should be significantly shorter than that for the `ArrayBoard` functions. If your solution is longer than 10 lines for any of them, you should rethink it.
- The OCaml set module functions (see here) will be extremely useful. Note that `LocSet` implements all of them. You don't need anything fancy (no unions, intersections, *etc.*).

Part B: The game functor

[120] Write a functor called `Game` that takes a module argument of module type `BOARD` and returns a module of module type `GAME`, which has this definition:

```
module type GAME =
  sig
    type t

    val play : t -> loc -> t
    val play_many : t -> loc list -> t
    val from_array : light array array -> t
  end
```

Here is what the contents of the `GAME` module type represent:

- The type `t` represents the game; this will be the same as the board type `t` of the `BOARD` functor argument.
- The `play` function takes a game (board) value and a `loc` and returns a new game (board) in which the location has been "played" as a move (the input argument is not altered). Note that "playing" a location means to flip that location in the board, as well as all the locations within one location in a horizontal or vertical direction as described above.
- The `play_many` function takes a game and a list of `loc`s and returns a new game where all the locations in the list have been played in sequence.
- The `from_array` function takes a 2-d array of `light`s and creates a game where the lights (which can be `On` or `Off`) are used to set the corresponding locations of the game board. If the 2-d array can't be converted to a game (because its dimensions are wrong), it raises a `Failure` exception using the `failwith` function. (We are assuming that the 2-d arrays of all Lights Out games are 5x5).

The code for the functor itself looks like this:

```
module Game (Board : BOARD) : GAME with type t = Board.t =
  struct
    type t = Board.t

    (* your code goes here *)
  end
```

We can create different game modules by supplying the `Board` module argument:

```
module ArrayGame = Game(ArrayBoard)
module SetGame = Game(SetBoard)
```

This is done in the test code. Here are some notes on the functions in this module:

- `play`: This function should be defined by using the `flip` function of the board module. Make sure you flip all the adjacent locations as well as the location itself. *Hint*: the `List.filter` function is your friend. This function shouldn't take more than 20 lines tops, and can be done in less.

- `play_many`: This function should be defined by using the `play` function. If you do this it's very easy to write and quite short.

- `from_array`: The input array contains values of the type `light` which is defined to be `On` or `Off` in the template file. Don't forget to check the dimensions of the input array. Don't assume that all rows will have the same lengths! Once this is done, iterate through all elements of the array and collect up the locations which are `On`. Then use this to create the board. Note that returning a "board" is the same as returning a "game", since they both have the same type. *Hint:* the `Array.length` and `Array.for_all` functions will probably be useful to you here.

The interactive game functor

In the template code, we are also supplying you with a functor call `Interact` which takes a board module and returns a module which allows you to play a game of Lights Out against the computer. Internally, it uses the board module to create a game module and then uses that along with some other functions it defines to allow interactive play.

There is nothing to hand in for this section, but you can run this code to test your code or just to have fun. Start the OCaml interactive interpreter:

```
$ ocaml
```

Once in the interpreter, type this:

```
# #use "lab7.ml";;
# test_array_board ();;  (* to test the array board implementation *)
# test_set_board ();;    (* to test the set board implementation *)
```

Either `test_` function will bring up an interactive interface which will allow you to play the Lights Out game. See above for a sample game. Lights Out is not a brutally difficult puzzle game, but it's not trivial either.