```
(*
                          CS 51 Lab 19
                Conway's Game of Life Cellular Automaton

Objective:

This lab provides practice with *refactoring a complex system* in
order to break up the complex system into separate simpler systems,
while generalizing them as well.
 *)

(*.....................................................................
Background

A cellular automaton (CA) is a model of computation that takes
place on a grid of cells each in a specific state. An "update rule"
specifies how each cell in the grid should be updated based on its
current state and the states of neighboring cells. The automaton
proceeds through "generations" or "ticks" in which all of the cells
are simnultaneously updated each to its new state.

Cellular automata have application in many areas as models for

  o biological processes like seashell patterns, tiger stripes, and
    giraffe reticulation;

  o chemical oscillators like the Belousov-Zhabotinsky reaction;

  o Ising models in physics;

  o artistic renderings and generative landscapes in video games;

  o and perhaps even *everything in the universe*, as in Stephen
    Wolfram's "New Kind of Science".

A particular cellular automaton you may be familiar with is John
Horton Conway's "Game of Life" (GoL). In this CA, cells are in one of
just two states: "alive" or "dead". The update rule for GoL is:

  o Cells that are dead become alive if they have exactly three live
    neighbors.

  o Cell that are alive stay alive only if they have two or three live
    neighbors.

Here, the "neighbors" of a cell are the eight cells that surround
it. (For cells along the edges of the grid, we think of their
neighbors as wrapping around, as if the grid were on a torus.)

As an example, consider this small grid of cells (a), where "-"
indicates a dead cell and "*" indicates a live one. According to the
GoL update rule, after one generation, the grid will update to (b),
and then to (c), and so on.

         -----        -----        -----
         --*--        -----        -----
         -*---        -*-*-        -*---
         -***-        -**--        -*-*-
         -----        --*--        -**--
          (a)          (b)          (c)

You can read more about the Game of Life at
<https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life>.

.....................................................................
```

The lab implementation of GoL

In the implementation of the GoL CA in this file, cell states are represented by boolean values (`true` = alive, `false` = dead) and the grid of cells as an array of array of booleans. You should be able to compile and run this code as is by doing:

```
% ocamlbuild -use-ocamlfind life.byte
% ./life.byte 3
```

You should see the automaton play out in an OCaml graphics window, as shown in the video at <https://url.cs51.io/lifevideo>.

[Note: In the `life_update` function we define below, as used in the video, we actually augment the original GoL update function by allowing for very occasional random changes of cell state. This keeps things interesting over many generations, so that the grid doesn't end up in repeating patterns as much.]

The implementation provided intermixes code for simulating and rendering CAs *in general*, and the particulars of Conway's GoL CA *in particular*. A more modular and general setup would factor these two facets apart, by providing an abstract data type for 2D cellular automata in general, which is the goal of this lab.

The `Automaton` module defined in the file `cellular.ml` is set up to implement cellular automata based on a specification of this sort. This ADT approach is much more general. For instance, in the ADT, cell states are taken to be values of an arbitrary type (not just `bool`).

****
Your job is to complete the implementation of the `Automaton` functor in that file and to refactor the code in this file to make good use of it. Undoubtedly, your refactoring will involve moving various bits of code from this file into `cellular.ml`, and making use of the `Automaton` functor in what remains in this file.
****

Once you've completed the refactoring, this file will continue to work as an implementation of the Game of Life, but as a bonus, you'll also be able to run several other cellular automata, showing the generality of the refactoring:

- greenbergHastings.ml
  The Greenberg-Hastings model
  https://en.wikipedia.org/wiki/Greenbergâ\200\223Hastings_cellular_automaton

- briansBrain.ml
  "Brian's Brain"
  https://en.wikipedia.org/wiki/Brian%27s_Brain

- reactionDiffusion.ml
  A simple reaction-diffusion model
  https://en.wikipedia.org/wiki/Reactionâ\200\223diffusion_system

Hint: You may want to refer to these files to get an idea of how your refactored code will work once you've got it together.

```
**********
WARNING: If you have photosensitive epilepsy, you should avoid
viewing the Greenberg-Hastings and reaction-diffusion cellular
automata, which exhibit rapidly flashing visual patterns.
********** *)
```

```
module G = Graphics ;;

(* Automaton parameters *)
let cGRID_SIZE = 100 ;;      (* width and height of grid in cells *)
let cSPARSITY = 5 ;;         (* inverse of proportion of cells initially live *)
let cRANDOMNESS = 0.00001 ;; (* probability of randomly modifying a cell *)

(* Rendering parameters *)
let cCOLOR_LIVE = G.rgb 93 46 70 ;;       (* color to depict live cells *)
let cCOLOR_DEAD = G.rgb 242 227 211 ;;    (* background color *)
let cCOLOR_LEGEND = G.rgb 173 106 108 ;;  (* color for textual legend *)
let cSIDE = 8 ;;             (* width and height of cells in pixels *)
let cRENDER_FREQUENCY = 1    (* how frequently grid is rendered (in ticks) *) ;;

(* Font specification for rendering the legend. OCaml font handling is
   platform dependent, so we leave this as `None`, but we provide the
   functionality for use in the solution set. Feel free to play with
   this or leave it as is. *)
let cFONT = None ;;

(* fresh_grid () -- Returns a grid with all cells in initial states *)
let fresh_grid () =
  Array.make_matrix cGRID_SIZE cGRID_SIZE false ;;

(* graphics_init () -- Initialize the graphics window to the correct
   size and other parameters. Auto-synchronizing is off, so our code
   is responsible for flushing to the screen upon rendering. *)
let graphics_init () =
  G.open_graph "";
  G.resize_window (cGRID_SIZE * cSIDE) (cGRID_SIZE * cSIDE);
  (match cFONT with
   | None -> ()
   | Some fontspec -> G.set_font fontspec);
  G.auto_synchronize false ;;

(* render_grid grid legend -- Renders the `grid` to the already
   initalized graphics window including the textual `legend` *)
let render_grid (grid : bool array array) (legend : string) : unit =
  (* draw the grid of cells *)
  for i = 0 to cGRID_SIZE - 1 do
    for j = 0 to cGRID_SIZE - 1 do
      if grid.(i).(j) then
        G.set_color cCOLOR_LIVE
      else
        G.set_color cCOLOR_DEAD;
      G.fill_rect (i * cSIDE) (j * cSIDE) (cSIDE - 1) (cSIDE - 1)
    done
  done;
  (* draw the legend *)
  G.moveto (cSIDE * 2) (cSIDE * 2);
  G.set_color cCOLOR_LEGEND;
  G.draw_string legend;
  (* flush to the screen *)
  G.synchronize () ;;

(* copy_grid src dst -- Destructively updates `dst` grid to have
   contents of `src` grid *)
let copy_grid src dst =
  for i = 0 to cGRID_SIZE - 1 do
    for j = 0 to cGRID_SIZE - 1 do
      dst.(i).(j) <- src.(i).(j)
    done
  done ;;

(* update_grid grid fn -- Applies `fn` to the grid and indices of each
```

```
   cell, updating the `grid` simultaneously with the returned
   values. Uses a static temporary grid so that all changes are
   simultaneous. *)
let update_grid =
  (* use a single static temp grid across invocations *)
  let temp_grid = fresh_grid () in
  fun (grid : bool array array)
      (fn : bool array array -> int -> int -> bool) ->
    for i = 0 to cGRID_SIZE - 1 do
      for j = 0 to cGRID_SIZE - 1 do
        temp_grid.(i).(j) <- fn grid i j
      done
    done;
    copy_grid temp_grid grid ;;

(* offset index off -- Returns the `index` offset by `off` allowing
   for wraparound. *)
let rec offset (index : int) (off : int) : int =
  if index + off < 0 then
    cGRID_SIZE - (offset ~-index ~-off)
  else
    (index + off) mod cGRID_SIZE ;;

(* life_update grid i j -- This CA update rule returns the updated
   value for the cell at `i, j` in the grid based on the rules of
   Conway's Game of Life. In this implementation, after updating as
   per the standard GoL update rule, a cell's state is then flipped
   with probability given by cRANDOMNESS. *)
let life_update (grid : bool array array) (i : int) (j : int) : bool =

  (* These lists are indexed by adjacency count to give the generated
     cell status.
                       0      1      2      3      4      5      6      7      8
                       |      |      |      |      |      |      |      |      |
                       v      v      v      v      v      v      v      v      v
   *)
  let live_update = [false; false;  true;  true; false; false; false; false; false] in
  let dead_update = [false; false; false;  true; false; false; false; false; false] in

  (* get count of adjacent live neighbors *)
  let neighbors = ref 0 in
  for i' = ~-1 to ~+1 do
    for j' = ~-1 to ~+1 do
      if i' <> 0 || j' <> 0 then
        let i_ind = offset i i' in
        let j_ind = offset j j' in
        neighbors := !neighbors
                     + if grid.(i_ind).(j_ind) then 1 else 0
    done
  done;
  (* determine new state based on neighbor count *)
  let new_state =
    if grid.(i).(j) then
      List.nth live_update !neighbors
    else
      List.nth dead_update !neighbors in
  (* randomly flip an occasional cell state *)
  if Random.float 1.0 <= cRANDOMNESS then
    not new_state
  else
    new_state ;;

(* random_grid count -- Returns a grid with cells set to live at
   `count` random locations. *)
let random_grid count =
```

```
  let mat = fresh_grid () in
  for _i = 1 to count do
    mat.(Random.int cGRID_SIZE).(Random.int cGRID_SIZE) <- true
  done;
  mat ;;

let main seed =
  Random.init seed;
  graphics_init ();
  let tick = ref 0 in
  let grid = random_grid (cGRID_SIZE * cGRID_SIZE / cSPARSITY) in
  while not (G.key_pressed ()) do
    if !tick mod cRENDER_FREQUENCY = 0 then
      render_grid grid (Printf.sprintf "Life: tick %d" !tick);
    update_grid grid life_update;
    tick := succ !tick;
  done;
  G.close_graph () ;;

(* Run the automaton with user-supplied seed *)
let _ =
  if Array.length Sys.argv <= 1 then
    Printf.printf "Usage: %s <seed>\n  where <seed> is integer seed\n"
                  Sys.argv.(0)
  else
    main (int_of_string Sys.argv.(1)) ;;
```