CMSC 330 Project #3 Fall 2012

Date due: Tuesday, November 6, 10:00:00 p.m.

1 Introduction

In this project you will write a variety of small OCaml functions, write a rational number module, and implement a computer science simulation called the Game of Life, which models or mimics some biological aspects of life including loneliness, congregation, and overcrowding. The purpose of the project is to get practice with OCaml data, recursion, and modules. With one exception, all functions to be written that have more than one argument will use currying, so a call to the function would simply be in the form of the function name followed by arguments (i.e., the arguments would not be in the form of a tuple.)

The project is divided into four parts. Each part must be written in a separate source file,named `part1.ml` through `part4.ml`. You may write helper function definitions in any of the files as you need, but do **not** include any top–level print calls or expression evaluations. (In other words, if we were to run OCaml and type for example `#use "part1.ml"`, we should only see definitions of functions.) If you include print calls or expression evaluations in any of the files their results will throw off the comparison of the output, and consequently your program will fail all of the tests of that part on the submit server. That would be unfortunate.

In several places, some functions are described as having to throw `Failure` exceptions (as mentioned in class, `Failure` is an OCaml library exception that takes one argument, which is commonly used for a description of the problem). In these cases it doesn't matter what argument string the exception is constructed with. Some of our tests may check that the right exception is being thrown in these cases, but we will not check its argument string's value. In the discussion below, "pair" refers to a 2–tuple.

2 Part #1: Small OCaml functions

Put all of the functions for this part in a single file `part1.ml`. All functions in this part that take individual numbers as arguments should take `int`s. For functions that take lists, in any cases where it matters the element types of the lists have been specified below; other functions that have list arguments may be called with any type of list. The first few functions are quite simple. Some of the functions, including the simple ones, may be used in writing other ones, or in our tests of other ones. As Section 8 says, you may **not** use any OCaml library functions in any of your functions. For a couple of the functions you also may not use any user–written helper functions, as specified below.

- Write a function `double x` that returns two times `x`'s value.
- Write a function `absolute_value x` that returns the absolute value of `x`.

  Note: if you write a call like `absolute_value -1` in OCaml the interpretation is that you are trying to subtract `1` from something named `absolute_value`, but `absolute_value` is a function, which you can't subtract an integer from. Instead, use `absolute_value (-1)`.
- Write a function `power_of_2 x` that returns true if and only if `x` is a power of 2. Use the following facts:
  - 1 is a power of 2
  - `x` is a power of 2 if 2 divides `x` and `x / 2` is a power of 2. Hint– you may want to use the `mod` function here, which returns the remainder of a division (like `%` in Java or C).

  If its parameter `x` is negative the function should throw a `Failure` exception.
- Write a function `chebyshev x k` that computes the function $T_k(\mathtt{x})$, which is defined as follows:

$$T_k(x) = \begin{cases} 1 & \text{if } k = 0 \\ x & \text{if } k = 1 \\ 2T_{k-1}(x) - T_{k-2}(x) & \text{if } k > 1 \end{cases}$$

  If its parameter `k` is negative the function should throw a `Failure` exception. (The definition of the function above shows that it is not a problem if `x` is negative.)

Important: The computation of `chebyshev x k` should take time $\mathcal{O}(\texttt{k})$. To achieve this, you will need to write a second function (you may want to name it `chebyshev' x y z k`; note that identifiers in OCaml can contain single quotes, so helper functions are frequently named with a trailing quote like this) where `y` and `z` are $T_{k-1}(\texttt{x})$ and $T_{k-2}(\texttt{x})$. Hint: You may also want to implement the straightforward version of `chebyshev` to check your results.

- Write a function `product list` that returns the product of the elements in the list `list`. (You may assume that its argument is a list of integers.) The function should **not** use any other user–defined (helper) functions. The product of the empty list is defined to be 1.
- Write a function `dotproduct list1 list2` that returns the dot product of the lists `list1` and `list2`. (You may assume that both arguments are lists of integers.) If you haven't had linear algebra, the dot product of two lists x and y, which each have n elements, is defined as $\sum_{i=1}^{n}(x_i \cdot y_i) = x_1y_1 + \cdots + x_ny_n$. For example, `dotproduct [1; 2; 3] [4; 5; 6]` should return 32. If the lists do not have the same length the function should throw a `Failure` exception.
- Write a function `remove_first x list` that returns a copy of the list `list` with the **first** occurrence (and only the first occurrence) of the element `x` removed. It should return `list` itself if `x` is not in the list. The function should **not** use any other user-defined helper functions.
- Write a function `smallest list` that returns the smallest element in the list `list`. (You may assume that its argument is a list of integers, although you may be surprised after you have written it to see that it will work with a variety of types anyway.) If the list is empty the function should throw the exception `Failure`. The function should not assume anything about the values in the list; there may be negative numbers in it, and there's no guarantee that everything in the list will be greater than some particular number.
- Write a function `selection_sort list` that returns a copy of the list `list` sorted from smallest to largest element, using the selection sort algorithm. (You may assume that its argument is a list of things that can be compared with comparison operators, and as above may be surprised after you have written it to see that it will work with a variety of types.)

  Consider the following idea: If the first element of the list is the smallest element, it will be at the beginning of the result. Otherwise, find the smallest element in the list, and make it the head of the result list. The functions `remove_first` and `smallest` may be useful in accomplishing this.
- Write a function `string_of_int_list list` that converts a list of integers to a string, using the `string_of_int` OCaml library function, and returns the string produced. The string returned should be exactly the same as if the list were printed by the top–level in OCaml, e.g., the last element should not have a trailing semicolon. Note that OCaml's string concatenation function is `^`. For example, the call `string_of_int_list [3; 3; 0]` should return the string (blank spaces are shown as ␣) "`[3;␣3;␣0]`" (without the quotes).
- Write a function `addpairs list` whose parameter `list` is a list of pairs of integers, and which returns a list containing the sum of each pair in the same order the pairs appeared in the list. For example, `addpairs [(1, 2); (3, 4); (5, 6)]` should return the list `[3; 7; 11]`.
- Write a function `flatten list` whose parameter `list` is a list of lists of integers, and which returns a single list containing all the integers in `list` in order, from the first list to the last. For example, `flatten [[1; 2; 3]; []; [4; 5]]` should return `[1; 2; 3; 4; 5]`. Note that the lengths of the inner lists may not be the same.

3 Part #2: Higher–order functions

Put all of the functions for this part in the file `part2.ml`. You may assume that all parameters in this part that are of a function type will be of type `int -> int`.

- Write a function `app_pair fn pair` that returns a new pair (as above, a two-tuple) containing `fn` applied to each component of `pair`, which you may assume is a pair. For example, `app_pair double (3, 4)` should return `(6, 8)`.
- Write a function `apply_n_times fn n x` that returns `fn(fn(...fn(x)))`, where the function `fn` is applied `n` times to `x`. `apply_n_times fn 0 x` is just `x`. If `n` is negative the function should throw a `Failure` exception.
- Write a function `apply_to_range fn m n` that returns a list containing the function `fn` applied to each of the integers from `m` to `n`, inclusive. For example, `apply_to_range double 3 5` should return `[6; 8; 10]`. If `m` is greater than `n` the function should throw a `Failure` exception.
- Write a function `compose list x`, where `list` is a list of functions, that returns the composition of all the functions in `list` applied to `x`. If `list` is empty, the result produced should just be `x`, which is just the identity function applied to `x`. For example, `compose [double; absolute_value] (-3)` = (mathematically) `double(absolute_value(`-3`))` $= 6$. Note the order of function application.

4 Part #3: A rational number module

In this part you will implement a module for working with rational numbers. In floating point, numbers such as $1/3$ cannot be represented exactly—the closest we can come is something like 0.333333, but eventually we run out of digits. In order to avoid rounding errors, we would like to keep the numerator and denominator separate, as integers. For example, you could represent a rational number n/d as the pair `(n, d)`. Your task is to implement a module `Rational` with the following signature:

```
module type RATIONAL =
  sig
    type t

    val rational : int -> int -> t          (* num -> den -> rat *)
    val string_of_rational : t -> string    (* num/den, reduced *)
    val numerator : t -> int                (* numerator *)
    val denominator : t -> int              (* denominator *)
    val greaterthan : t -> t -> bool        (* true if 1st arg > 2nd arg *)
    val multiply : t -> t -> t              (* multiplication *)
    val add : t -> t -> t                   (* addition *)
    val ceiling : t -> int                  (* ceiling *)
  end
```

Here `t` is an abstract type representing rational numbers. A client using this module will not be able to assume anything about how rationals are represented. Here is a description of the functions' behavior:

`rational:` A more conventional OCaml name for this function, to follow the naming convention used by other functions that convert one type to another, would be `string_of_rational`, but we named it `rational` because it's sort of like a constructor, and in languages that have been taught up to this point a constructor would have the name of its class or type. It should produce a rational number from two integers, whose values may both be positive, both be negative, or one may be positive and the other negative (or either or both may be zero). Note that all functions should work appropriately, without errors, with fractions having the value zero.

`string_of_rational:` This function should convert a rational to a string representation `n/d`, where `n` is the numerator and `d` is the denominator. Negative rational numbers should be returned as `-n/d`, and never as `n/-d`. It must always return a string representing the rational in **reduced** form, so that the greatest common divisor (gcd) of `n` and `d` is 1. To achieve this you will probably need to write a function to compute the gcd. To calculate the gcd of two numbers you can use the naive algorithm (dividing both numbers by all possible numbers that could be the gcd, and returning the largest), or you can use Euclid's algorithm, one form of which is:

```
gcd(a, b):
  if a == 0
    return b
  else
    while b != 0
      if a > b
        a= a - b
      else b= b - a
    return a
```

There is also a modulus–based version of Euclid's algorithm, which you can look up and use if you prefer.

`numerator`: This function should return the numerator of the rational it is applied to, also in reduced form. `numerator` of a negative rational should always be negative.

`denominator`: This function should return the denominator of the rational it is applied to, also in reduced form. `denominator` of a negative rational should be **positive**.

`greaterthan`: This function should return true if the value of its first rational argument is (strictly) greater than the value of its second rational argument.

`multiply`: This function should return a new rational representing the product of its two rational arguments.

`add`: This function should return a new rational representing the sum of its two rational arguments.

`ceiling`: This function should return the smallest integer greater than or equal to its rational argument. (E.g., ceiling(2/3) is 1, and ceiling(−2/3) is 0.)

Handling negative rationals may unexpectedly be more tricky than handling positive rationals. You may want to just get the functions to work at first with positive rationals, then modify them.

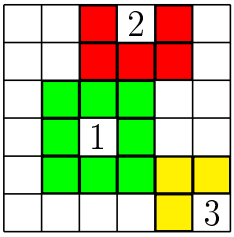
Put your solution for this part in a file `part3.ml`. Don't create separate `.mli` and `.ml` files; just put both the module signature and implementation into the same file `part3.ml`.

5 Part #4: Implement the Game of Life

The Game of Life was originally invented by John H. Conway, a mathematician at Cambridge, and described by Martin Gardner in his "Mathematical Games" column in the October 1970 issue of *Scientific American*. Although it's called a game, Life is an example of a cellular automaton; these have applicability in simulations of physics, chemistry, and biology, as well as studying computer viruses, and they're used in image processing and image generation. A 1997 Scientific American article mentioned using cellular automata to simulate and study solutions to traffic problems.

Life occurs on a two–dimensional rectangular grid divided into different cells, and takes place over one or more generations. Each cell in the grid may be either empty or occupied by a single creature. The cells that will contain creatures in any generation are determined according to how many neighbors each has in the previous generation. The rules in Conway's original version of the game that govern how creatures multiply and decrease are quite simple; for this project, in order to be able to experiment with different situations, we will allow the rules to be varied during any particular execution.

In order to explain the game rules we need to describe what the neighbors of a cell are, namely all the cells that are adjacent to it. Note that a cell is not a neighbor of itself. Not all cells in a grid have the same number of neighbors. An interior cell, like the one marked 1 below, has eight neighbors (any, all, or none of which may be occupied by a creature), which are the ones shaded in green.

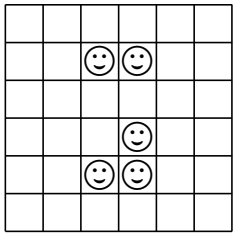


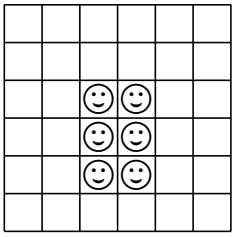
An edge cell, like the one marked 2 above, has five neighbors; the ones shaded in red, and a corner cell, like the one marked 3 above, has three neighbors, which are the ones shaded in yellow. Your program will have to examine all of the neighbors of each cell and count how many are occupied by creatures, to be able to implement rules that determine what the next generation will be.

In Conway's original version the creatures' population changes according to the following rules:

- An empty cell with exactly three neighboring creatures will undergo a "birth", meaning it will contain a creature in the next generation, simulating that a certain population density is necessary for population growth and reproduction, but conditions cannot be too crowded or a species will not expand.
- An empty cell with fewer than three or with more than three neighboring creatures will remain empty in the next generation.
- The creature in an occupied cell will continue to live for another generation only if it has either two or three neighboring creatures.
- A creature with fewer than two neighboring creatures will not survive in the next generation (its cell will become empty), simulating the effect of loneliness on population and modeling the situation where a minimum population density is necessary for individuals of a species to survive.
- Any creature with more than three neighboring creatures will also not survive in the next generation, simulating the effects of overcrowding and contention for scarce resources on a population.

For example, if ☺ represents a creature, the arrangement of creatures in the grid on the left below gives rise to the succeeding generation shown on the right:

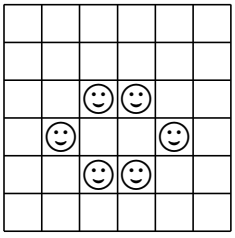




The two creatures in the second row did not survive to the next generation, as they only had one neighbor apiece (each had the other). There were two births in the third row, as the two center cells in that row each had three neighbors: the two in the second row (one directly above and the other diagonally above), and the one in the fourth row below (directly below in one case, and diagonally below in the other). There was one birth in the fourth row, and the three remaining creatures remained alive in the next generation as they each had two neighbors. Note that cells not on the grid can be considered to be dead neighbors.

The Game of Life gives rise to a number of interesting possibilities; in particular substantial variations in eventual results can arise from even minor differences in the starting conditions:

- Some initial arrangements of creatures on the grid are examples of stasis, meaning they are patterns that stay exactly the same when the rules governing reproduction and death are applied, so every generation is exactly the same as the initial generation or starting arrangement of creatures.
- Some starting arrangements lead to all of the creatures dying within some number of generations.
- Some starting arrangements lead to patterns that move across the grid (called gliders).
- With some starting arrangements the population of creatures changes for some number of generations, but eventually reaches stasis and doesn't change thereafter. For instance, consider the example above– the next or third generation, as well as all subsequent ones, would look like this:



It can be fascinating to see how some very simple initial arrangements of creatures can lead to the development of complex patterns of reproduction.

Although your program will be able to simulate Life using Conway's original rules above, we will allow for variation in the rules by permitting the numbers of neighbors that govern the birth and death of creatures to be supplied when the game begins. There are actually tens of thousands of possible variations on the game rules, that yield very different behavior in some cases.

In this project, grids of cells will be represented by data of the following type:

```
type board = int * int * int list list
```

Here a board is a tuple `(max_row, max_col, ll)` where `max_row` is the number of rows of the board, `max_col` is the number of columns in the board, and `ll` is a list of lists of integers containing the cells of the board in order by row (i.e., row–major order). The integer 1 represents a cell with a living creature, and the integer 0 represents an empty cell. For example, here is the definition of one sample board:

```
let sample =
  (5, 11,
    [[0;0;0;0;0;1;1;0;0;0;0];
     [0;0;0;0;1;1;0;0;0;0;0];
     [0;0;0;0;0;1;0;0;0;1;0];
     [0;0;0;0;0;0;0;0;0;1;0];
     [0;0;0;0;0;0;0;0;0;1;0]])
```

The two integers `max_row` and `max_col` are actually not necessary, because they could be computed from the length of the list and the lengths of the inner list, but they are just to make things easy and avoid having to count.

In addition to the playing board, your game simulation will take as input a function specifying the rules of the game. This function will have the following type: `type rule = int * int list -> int`. The rule function takes as input a pair `(cell, adjacent)`, where `cell` is an integer representing the state of the cell in question (1 for alive, meaning containing a creature, or 0 for empty), and `adjacent` is a list containing the states of the surrounding cells. It should return either 0 or 1, depending on whether in the next generation the cell will be empty or will have a creature, respectively. As above, if any neighbors of a cell don't exist because the cell is an edge cell, the game behaves as if those neighbors existed and were empty.

You must define the following functions, which must be in a file `part4.ml`:

- A function `standard_rule : rule` that implements Conway's standard rules described above, and
- A function `generate` that takes two arguments as input (not a tuple of arguments), a board and a rule, and runs one step of the game on that board using that rule.

You may assume that the board passed in to the `generate` function is well–formed (i.e., the list elements are either 0 or 1, the lengths of the inner lists are the same, and the number of the inner lists and their lengths agree with the first two numbers. `generate` may behave arbitrarily and produce any result, including raising exceptions, if its board is not well–formed. You may assume the rule argument is a function that returns either 0 or 1. You may assume– although it is not actually necessary– that any rule function will be called with a list containing exactly eight elements. Note that the `generate` function is a higher–order function, since it takes a rule function and applies it to the board, returning the board that results.

(Note that the secret tests may use other rules besides the standard rule, so you may want to write other rules and test your `generate` function with them.)

6 Example

The example below shows an interactive session with OCaml; the user's input is entered after each OCaml prompt. The UNIX prompt on this system is `grace2:~/330/proj3:`. For clarity the user's input is shown in blue, while everything else is output produced by the OCaml top–level.

```
grace2:~330/proj3: ocaml
        Objective Caml version 3.12.1

# #load "utilities.cmo";;
# #use "part1.ml";;
val double : int -> int = <fun>
val absolute_value : int -> int = <fun>
val power_of_2 : int -> bool = <fun>
val chebyshev : int -> int -> int = <fun>
val product : int list -> int = <fun>
val dotproduct : int list -> int list -> int = <fun>
val remove_first : 'a -> 'a list -> 'a list = <fun>
val smallest : 'a list -> 'a = <fun>
val selection_sort : 'a list -> 'a list = <fun>
val string_of_int_list : int list -> string = <fun>
val addpairs : (int * int) list -> int list = <fun>
val flatten : 'a list list -> 'a list = <fun>
# double 330;;
- : int = 660
# power_of_2 1048576;;
- : bool = true
# power_of_2 1048575;;
- : bool = false
# product [3; 1; 4; 1; 5; 6];;
- : int = 360
```

Pressing control–d stops (quits) the OCaml top–level.

7 Test utilities and public tests

The previous section showed an interactive session with the OCaml top level, which is probably how you will develop your program. This won't work on the submit server, so the public tests will work a little differently.

For each type of value that could be returned by a function we wrote a function that prints that kind of value, for example, `print_bool` prints a boolean value, `print_int_list` prints a list of integers, `print_char_list`, prints a list of characters, etc. They all print their argument followed by a newline, using OCaml's `print_endline` function. (The only exception is that we didn't write a function to print a string followed by a newline, because `print_endline` already does that.) These functions are supplied in the compiled OCaml object file `utilities.cmo`. The public test "inputs" are OCaml source files (there are no files that input should be redirected from), and we use these functions in the public tests to print the results returned by your functions. Note that it's necessary to use OCaml's `open` to access the functions in this file. Each public test file first loads one of your four source files, then loads the utilities, then prints the results of some calls to your functions. These input files may also contain definitions of variables that are used in later lines.

For example, a hypothetical public test that tests only two calls to your functions could look like the following (suppose it's in a file named `public0.ml`):

```
#use "part1.ml";;
#load "utilities.cmo";;

open Utilities;;

print_int  (double 330);;
print_bool (power_of_2 1048576);;
```

It would be run as `ocaml public0.ml`. For this example, the output file would just contain two lines, the first with the number 660 and the second with the word true, both ending in newlines.

The tests of the third part are the same, except that since the rational number module is in fact a module, it must be opened before the functions can be used. Note that if you are testing the third part in the OCaml top level, change your source file and re–load `part3.ml`, you will **also** have to re–`open Rational`, otherwise the top–level will still be using the **old** version of the rational module.

7.1 Testing the fourth part

To test your functions for the Game of Life you can use the routine `run` that we wrote in the utilities module in `utilities.cmo` to run the simulation indefinitely, printing text output, using your functions above. It has three arguments: a board, the generate function, and a rule function. Every time you press the return or enter key, the next generation is displayed. For example, try `run sample_board generate standard_rule` to run the game on the sample board that we defined in Section 5 (the sample board is predefined in the utilities module). Use control–c to stop execution. Note that this is also a higher–order function– it takes a board, your `generate` function, and a rule function (such as your `standard_rule` function, although another rule function could be written and used), and repeatedly applies the two functions to the board, printing the results.

Using this `run` function is not going to work when projects are submitted to the submit server, because there is no one on the submit server to keep pressing enter. Instead, any tests of the fourth part will use a function (also in `utilities.cmo`) `run_n_times` that runs the Game of Life for a certain number of generations, and prints the result at the end of that number of generations. It has four arguments: the same three arguments as `run` does, plus an integer specifying how many generations to run the simulation for. For example, `run_n_times sample_board generate standard_rule 4` would run the simulation for four generations on the sample board, using your `generate` and `standard_rule` functions.

These functions work, but their output can be a little hard to read. You can also use a function `run_graphical` that will draw a picture of the board in a window. This will work for boards up to around 20 by 20; for larger boards the bottom will be cut off.

For the `run_graphical` function to work you'll need to have an X–windows server running locally. For example, if you're logging in from a Windows machine you can install and use Xming (you should also install the fonts package), and if you're logging in from a Mac you can install and use XQuartz. If you're using a Linux machine you're presumably already running an X server, but you may need to log into the Grace machines using `ssh -X` or `ssh -Y`. (If you're currently able to run Emacs on the Grace machines and have it pop up in its own window, you don't need to install anything or to do anything differently).

The `run_graphical` function is in the compiled file `utilities2.cmo`. To use it, you will first have to load the graphics library and open the Graphics module. So a session using this function would look like:

```
#load "graphics.cma";;
open Graphics;;
#load "utilities2.cmo";;
open Utilities2;;
#use "part4.ml";;
run_graphical sample_board generate standard_rule;;
```

(A `.cma` OCaml file is an archive or library, similar to a static library in C, discussed in CMSC 216.) To run the graphical simulation, move the cursor into the OCaml graphics window that pops up, and every time you press any key the next generation is generated and appears. Move cursor back into the window where you're running the OCaml top–level and type control–c to stop it running. (Of course you may want to define other boards and run the simulation on them.)

If you want to try an alternative rule, the following produces an interesting result with the sample board: If a cell is dead and has exactly 2 or 3 live neighbors, then the cell becomes live, otherwise it dies or remains dead.

8 Project requirements and submitting your project

1. You **may not use any OCaml library modules**, for example, anything with a module name and a dot in front of it, such as `List.length`, `List.append`, `List.rev`, etc., may not be used. Built–in types, library exceptions, and the automatically–loaded functions in the `Pervasives` module (which include all the mathematical and logical operators, and any functions mentioned above that you may need, like string concatenation), may be freely used.

2. References and loops (OCaml does actually have loops) **may not be used at all anywhere**.

3. You can create helper functions to be called by your other functions if you like (other than for the `product` and `remove_first` functions. For your own helper functions you may use currying, or tuples as arguments, as you like.

4. We will be checking your source code to ensure that your implementation of the function `chebyshev` is $O(\texttt{k})$, that your functions don't call any library module functions, that your function `product` and `remove_first` don't call any helper functions, and that no references or loops are used. Violation of these requirements may result in a very large grading penalty.

5. Your code **cannot produce any incomplete match warnings**, or they will throw off the comparison of the output and you will fail all the tests of the affected parts. If any of your functions do not need to work in some match cases (based on the definitions of the functions above), just have them raise `Failure` exceptions in those cases, which will eliminate the match warnings.

6. As mentioned earlier in class, in OCaml the `==` operator tests for physical equality and `=` tests for structural equality. In any cases where you have to compare things for equality you should use `=`.

7. Your submitted functions **must** be in the four files named `part1.ml`, `part2.ml`, `part3.ml`, `part4.ml`, otherwise the submit server may not recognize them.

8. To check that your output matches the expected output you can use the UNIX `diff` command, for example:

   ```
   ocaml public1.ml > my-output
   diff -u public1.output my-output
   ```

   (Note that the output should compare **exactly**, including whitespace so do **not** use diff's `bB` options.)

   If no output at all is produced by `diff`, your output matches the expected output and is correct. This `diff` command is exactly what the submit server is going to be using to check your program's output, so using it you can see yourself whether your output is right or not; if you don't get matching results when comparing your output this way then the submit server is not going say your results are right either.

9. Each of your source files **must have** a comment near the top that contains your name, TerpConnect login ID (which is your directory ID), your university ID number, and your section number.

   The Campus Senate has adopted a policy asking students to include the following statement on each major assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently you're requested to include this pledge in a comment near the top of your program source file. See the next section for important information regarding academic integrity.

10. Note that although you are not being graded on your source code or style (see the separate project grading handout), if the TAs cannot read or understand your code they cannot help should you have to come to office hours, until you return with a well–written and clear program.

11. As before, to submit your program just type the single command `submit` from the `proj3` directory that you copied above, where your program files `part1.ml` through `part4.ml` should be. In this project also you may lose credit for having more than fifteen submissions.

9 Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.