Homework 8: Matrices
Due: 5:00 PM, Nov 1, 2017

Contents

1 Reverse (Practice) 3
2 Main Diagonal (Practice) 4
3 Horizontal Flip 4
4 Vertical Flip 5
5 Transpose 7
6 Well-ordering proofs 7

Objectives

By the end of this homework you will be able to:

1. write procedures in OCaml
2. transpose a matrix

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Horizontal Flip, Vertical Flip, Transpose, and Well-Ordering Proofs questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem, as follows (e.g., transpose.ml corresponds to Transpose):

- README.txt
- CS17setup.ml
- horz_flip.ml
- vert_flip.ml
- transpose.ml
- proof.tex

For this assignment, all files you turn in that contain code must be OCaml files, so they must end with extension .ml. Your solution to the analysis problem should be \LaTeX-generated .tex and .pdf files.

**Note:** If you are using an online \LaTeX tool to write your proof, make sure that you try compiling your file using the terminal once you have copied it to your course directory. Instructions on how to do this can be found in Lab 6.

If you are using a departmental linux system, all your solution files should reside in your 
`~/course/cs0170/homeworks/hw08` directory.

For this and every assignment, you should also have a README.txt file whose first line contains only your CS-department email address, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

```
README.txt:
jfh@cs.brown.edu
There’s nothing to say except that I’m turning in these files plus this README
the way the instructions say that I should.
```

To hand in your solutions to these problems, you must zip your hw08 directory into a file hw08.zip. Hand in this zip file using the method you learned in the first lab: visit [https://tinyurl.com/cs0170-handin](https://tinyurl.com/cs0170-handin) to get started.

**The List Module**

In this homework, you will represent matrices using lists. Not surprisingly, OCaml has many built-in procedures that operate on lists. Many of these procedures will be familiar to you from Racket, like List.hd, List.tl, and List.map. You may use any built-in procedures that correspond to ones we have written in Racket.

For a full list of procedures in OCaml’s List module, see [http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html](http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html)

**Note:** Remember to use the design recipe throughout this (and all) assignments!

**Matrices**

A matrix is a bunch of numbers arranged in a two-dimensional grid: e.g.,

\[
\begin{pmatrix}
1 & 2 & 3 \\
2 & 0 & -1
\end{pmatrix}
\]
We only call such a thing a matrix if it consists of at least one number; empty matrices are not allowed.

One natural way to represent a matrix is as a nonempty list of nonempty lists of numbers, where each list represents a row of the matrix. In this representation, you can’t have a matrix with no rows at all, and you can’t have a matrix of 5 rows and no columns, because each of these fails one of the two conditions.

Using this "list of rows" representation, the matrix above can be represented as follows:

\[
\begin{bmatrix}
[1; 2; 3]; [2; 0; -1]
\end{bmatrix}
\]

In a matrix, all rows (i.e., all inner lists) must be of the same length. However, the number of rows (i.e., the number of inner lists) and the number of columns (i.e., the length of each inner list) need not be equal. When they are equal, the matrix is said to be square. Otherwise, it is rectangular.

In a matrix \( A \), the element in the \( i \)th row and \( j \)th column is called \( a_{ij} \).

The following are examples of rectangular matrices, represented in both mathematical notation and in OCaml:

\[
\begin{bmatrix}
10 & 11 & 12 & 13 & 14 \\
15 & 16 & 17 & 18 & 19 \\
20 & 21 & 22 & 23 & 24
\end{bmatrix}
\]

\[
[[10; 11; 12; 13; 14]; [15; 16; 17; 18; 19]; [20; 21; 22; 23; 24]]
\]

\[
\begin{bmatrix}
10 & 11 & 12 \\
13 & 14 & 15 \\
16 & 17 & 18 \\
19 & 20 & 21 \\
22 & 23 & 24
\end{bmatrix}
\]

\[
[[10; 11; 12]; [13; 14; 15]; [16; 17; 18]; [19; 20; 21]; [22; 23; 24]]
\]

The procedures you write in this assignment should operate on both square and rectangular matrices, unless otherwise noted.

The goals of this homework are (1) to give you even more practice with recursion, this time in OCaml, (2) to give you a chance to show how well you can use higher-order procedures, and (3) to make you think hard about how to avoid relying on the behavior of procedures outside the domain of those procedures.

In particular, if we ask you to write a procedure \( \text{frink} \), that consumes and produces matrices, and you choose to write \( \text{frink} \) recursively, perhaps peeling off a row at a time, your base case should not be a matrix with no rows, i.e., an empty list. Why not? Because an empty list is not a matrix, and hence is not in the domain of \( \text{frink} \).

If you’re using \( \text{match} \) in your recursive procedure, as you should, you’ll still need to \( \text{match} \) the input that consists of an empty list, but that’s only to suppress Ocaml’s “Incomplete match”
error. A really good choice for a “result” in this match-case is a failwith expression, perhaps failwith "Domain error". Think about this carefully as you work on the “Main Diagonal” problem, and then read the discussion following the “Horizontal Flip” problem to see whether you worked things out properly.

Practice

1 Reverse (Practice)

Here’s the racket code for reverse from class.

\[
\text{(define (reverse alod)}
\text{(reverse-helper alod empty))}
\]

\[
\text{(define (reverse-helper alod part)}
\text{(cond}
\text{[(empty? alod) part]}
\text{[(cons? alod) (reverse-helper (rest alod) (cons (first alod) part))]])})
\]

Task: Rewrite this procedure in OCaml, using the Racket-to-OCaml cheat sheet.

Here’s a template to get you started:

\[
\text{let reverse (alod : 'a list) : 'a list = ...}
\]

\[
\text{let rec reverse_helper (alod : 'a list) (part : 'a list) : 'a list =}
\text{match alod with}
\text{| [] -> ...}
\text{| ... -> ...}
\]

Note: Make sure you use let rec and not let in defining reverse_helper

2 Main Diagonal (Practice)

The main diagonal of a square matrix is the diagonal that runs from the upper left hand corner to the lower right. More generally, the main diagonal of any rectangular matrix are the entries for which the row index, \(i\) equals the column index, \(j\). For example, the following matrix has 1s down its main diagonal:

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix}
\]

Task: Write a procedure main_diagonal that consumes a matrix and produces a list representing the main diagonal of the matrix.

For example,
main_diagonal \[
\begin{bmatrix}
10 & 11 & 12 & 13 & 14 \\
15 & 16 & 17 & 18 & 19 \\
20 & 21 & 22 & 23 & 24 \\
\end{bmatrix}
\] => \[
\begin{bmatrix}
10 & 16 & 22 \\
\end{bmatrix}
\]

**Hint:** One elegant solution to this problem makes use of the `List.map` procedure.

**Problems**

### 3 Horizontal Flip

We define the horizontal flip of a matrix as the matrix that results from reversing the order of the rows in the matrix. For example,

\[
\begin{pmatrix}
a & b & c \\
d & e & f \\
g & h & i \\
\end{pmatrix} \mapsto \begin{pmatrix}
g & h & i \\
d & e & f \\
a & b & c \\
\end{pmatrix}
\]

Note that if the matrix has \(n\) rows and \(k\) columns, then the horizontally flipped matrix also has \(n\) rows and \(k\) columns.

**Task:** Write a procedure `horz_flip` that takes as input a matrix and outputs the horizontal flip of that matrix.

**Note:** Matrices may not be empty, so the empty matrix is outside the domain of the function. You need not test a 0-dimensional matrix.

### 4 Vertical Flip

We define the **vertical flip** of a matrix as the matrix that results from reversing the order of the columns in the matrix. For example,

\[
\begin{pmatrix}
a & b & c \\
d & e & f \\
g & h & i \\
\end{pmatrix} \mapsto \begin{pmatrix}
c & b & a \\
f & e & d \\
i & h & g \\
\end{pmatrix}
\]

**Task:** Write a procedure `vert_flip` that takes as input a matrix and returns the vertical flip of that matrix. Use `List.map` to solve this problem.

We asked that you use `map` to solve the problem, but let’s imagine that you chose to use recursion instead. You’d create a recursive diagram like this, being a little informal about all the semicolons that should be in a matrix representation:

input:

```
[ [ a b c ]
  [ d e f ]
  [ g h i ]]
```
recursive input:
  [ [ d e f ]
    [ g h i ]]
recursive output:
  [ [ f e d ]
    [ i h g ]]

IDEAS:

overall output:
  [ [ c b a ]
    [ f e d ]
    [ i h g ]]

and in the “IDEAS” space, you might say something like “reverse the first row? Cons that onto
recursive result?”, and you’d find that this works just fine, so you’d write

```
let rec vert_flip( mat : 'a matrix) : 'a matrix =
  match mat with
  | [] => []
  | row::remainder -> (List.rev row):: (vert_flip remainder) ;;
```

This code works, but it is, from a CS17 perspective, incorrect. A procedure — even the one you’re
writing at any given moment — should never be applied to an argument that’s not in its domain.
And in vert-flipping our example matrix, we recursively call vert_flip on a matrix containing its
last two rows, and then on one containing its last row, and everything is OK up until now. But
finally, we call vert_flip on a “matrix” containing no rows at all — the empty list. And because
that’s not a matrix, we’ve done something wrong.

The revised version looks like this:

```
let rec vert_flip( mat : 'a matrix) : 'a matrix =
  match mat with
  | [] => failwith "Cannot vert_flip an empty matrix"
  | [row] => [List.rev row]
  | row::remainder -> (List.rev row):: (vert_flip remainder) ;;
```

And that is a valid recursive implementation of vert_flip, one that meets the standards of the
CS17 coding style guide.

By the way, this:

```
let rec vert_flip( mat : 'a matrix) : 'a matrix =
  match mat with
  | [] => []
  | [row] => [List.rev row]
  | row::remainder -> (List.rev row):: (vert_flip remainder) ;;
<several tests involving non-empty matrices here>
```
is also a valid implementation that meets the CS17 coding standard, because although the procedure happens to compute an answer for the empty list, it never uses that answer: the base case for recursion on a legal matrix is the second match-case.

Even though it’s a valid implementation, it’s not a particularly wise one: it’s easy in situations like this to forget that “nonempty” is part of the definition of matrix, and by “handling” the empty-list case, you’ve pushed the detection of the problem off to some other part of your program that might call vert_flip. This is a terrible thing, for what you want, when you’re debugging, is that errors appear exactly where they first occur.

“What if I’m doing something where some builtin does exactly what I need, but it handles the empty list, too? Do I have to write a match case and write

```ocaml
let my_proc( mat : 'a matrix) : 'a matrix =
  match mat with
  | [] -> failwith "Domain error!"
  | [row] -> some_builtin mat ;;
<several tests involving non-empty matrices here>
```

instead of

```ocaml
let my_proc( mat : 'a matrix) : 'a matrix = some_builtin mat ;;
```

in my code?”, you might ask. The answer is that you don’t have to, but as a safeguard against my own future mistakes, if I were writing the code, that’s what I would do.

## 5 Transpose

The transpose of a matrix is another matrix that results from reflecting the original matrix across its main diagonal. That is, the transpose $A^T$ of a matrix $A$ is defined as follows: for all positions $i, j$, $a_{ij} = a_{ji}$, meaning the element in the $i$th row and $j$th column of the transpose of a matrix is the same as the element in the $j$th row and $i$th column of the original matrix.

For example,

\[
\begin{pmatrix}
a & b & c \\
d & e & f \\
g & h & i \\
\end{pmatrix}
\mapsto
\begin{pmatrix}
a & d & g \\
b & e & h \\
c & f & i \\
\end{pmatrix}
\]

Note that transposing a non-square matrix swaps the number of rows and columns:

\[
\begin{pmatrix}
a & b & c \\
d & e & f \\
\end{pmatrix}
\mapsto
\begin{pmatrix}
a & d \\
b & e \\
c & f \\
\end{pmatrix}
\]

**Task:** Write a procedure `transpose` that consumes a matrix $A$ and produces $A^T$.

**Hint:** Here is a partial template for your solution. Note that it relies on the type definition of 'a matrix, which you must provide.
let rec transpose (mat : 'a matrix) : 'a matrix =
  match mat with
  | [] | [] :: _ -> failwith "A matrix cannot be 0-dimensional."
  | (hd1 :: []) :: tl -> (* TODO: base case: list of one-element lists *)
  | (hd1 :: tl1) :: tl -> (* TODO: recursive case: list of longer lists *)
  ;;

The first two patterns (the empty list, and [[[]]]) are not valid matrices. We include them nonetheless, because they are valid lists, and OCaml complains when pattern matching is not exhaustive.

6 Well-ordering proofs

Suppose that $H$ is a function on the positive natural numbers that satisfies the following recurrence:

$$
H(0) = A
$$

$$
H(n) \leq 6Bn^2 + H(n-1) \quad \text{for } n > 0
$$

Spike's bulldog Circe tried plug-n-chug on that recurrence and came up with the conjecture that

$$
H(n) \leq A + Bn(n+1)(2n+1) \quad \text{for } n \in \mathbb{N}
$$

(Circe’s algebra skills are pretty strong!)

**Task:** Construct a well-ordering proof that Circe’s conjecture is correct, by following the models from class.

**Note:** Proofs must be written in valid LaTeX. For a refresher on how to do this, reread Lab 6.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)