Homework 7: Matrices
Due: 10:59 PM, Oct 23, 2019

Contents

1 Reverse (Practice) 4
2 Main Diagonal (Practice) 4
3 Horizontal Flip (7 points) 5
4 Vertical Flip (7 points) 5
5 Transpose (11 points) 7
6 Well-Ordering Proofs (8 points) 8

Objectives

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

1. Write procedures in ReasonML
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. Make sure that your ReasonML files start with a capital letter. In particular, each should be named for the corresponding problem, as follows:

- README.txt
- CS17setup.re
- HorzFlip.re
- VertFlip.re
- Transpose.re
For this and every future assignment, you should also have a README.txt file whose first line contains only your Banner ID, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:
B01234567
There’s nothing to say except that I’m turning in some files plus this README the way the instructions say that I should.

To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them. If you re-submit your homework, you must re-submit all files.

The List Module

In this homework, you’ll represent matrices using lists. Not surprisingly, ReasonML 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, List.rev, and List.map. You may use any built-in procedures that correspond to ones we’ve written in Racket.

For a full list of procedures in ReasonML’s List module, see https://reasonml.github.io/api/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:

[[1, 2, 3], [2, 0, -1]]

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 ReasonML:

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

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

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

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

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 ReasonML, (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 `frink`, that consumes and produces matrices, and you choose to write `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 `frink`. If you're using `switch` in your recursive procedure, as you should, you'll still need to `switch` the input that consists of an empty list, but that's only to suppress ReasonML's "Incomplete switch" error. A really good choice for a "result" in this switch-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.

**Check Expects**

ReasonML is very particular about types. Because of this, check expects are going to be a little bit different for Reason. For this homework, when testing a procedure that outputs a `list('a)`, use `check_expect_list_alpha`. When testing a procedure that outputs a `list(list('a))`, use `check_expect_list_list_alpha`. Both of these procedures take THREE arguments: the actual value (the call to the function you're testing), the expected value, and a message (string) that indicates what that test is testing. These messages are to help you differentiate test outputs from one another; we will not be grading you on your test messages.
Example:

check_expect_list_alpha(reverse([]), [], "empty list")
check_expect_list_alpha(reverse([1]), [1] "one element list")

Output looks like:
ce_Success: empty list
ce_Success: one element list

1 Reverse (Practice)

Here’s the racket code for reverse from class.

```
(define (reverse alod)
  (reverse-helper alod empty))

(define (reverse-helper alod part)
  (cond
   [(empty? alod) part]
   [(cons? alod) (reverse-helper (rest alod) (cons (first alod) part))]))
```

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

Here’s a template to get you started:

```
let rec reverseHelper: (list('a), list('a)) => list('a) = (alod, part) =>
  switch (alod) {
    | [] => ...
    | [hd, ... tl] => ...
  };

let reverse: list('a) => list('a) = alod => ...;
```

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

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:

```
1 0 0 0
0 1 0 0
0 0 1 0
```

Task: Write a procedure mainDiagonal that consumes a matrix and produces a list representing the main diagonal of the matrix.
For example,

\[
\text{mainDiagonal}([[10, 11, 12, 13, 14], [15, 16, 17, 18, 19], [20, 21, 22, 23, 24]]) => [10, 16, 22])
\]

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

### 3 Horizontal Flip (7 points)

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 `horzFlip` 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 (7 points)

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 `vertFlip` 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 commas that *should* be in a matrix representation:

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

```ocaml
let rec vertFlip: matrix('a) => matrix('a) = mat =>
 switch (mat) {
 | [] => []
 | [row, ...remainder] => [List.rev(row), ...vertFlip(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 `vertFlip` 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 `vertFlip` 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:

```ocaml
let rec vertFlip: matrix('a) => matrix('a) = mat =>
 switch (mat) {
 | [] => failwith("Cannot vertFlip an empty matrix")
 | [row] => [List.rev(row)]
 | [row, ...remainder] => [List.rev(row), ...vertFlip(remainder)]
}
```

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

By the way, this:

```ocaml
let rec vertFlip: matrix('a) => matrix('a) = mat =>
 switch (mat) {
 | [] => []
 | [row] => [List.rev(row)]
 | [row, ...remainder] => [List.rev(row), ...vertFlip(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 switch-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 vertFlip. 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 switch case and write

```ocaml
let myProc: list('a) => list('a) = mat =>
  switch (mat) {
    | [] => failwith("Domain error!")
    | [row] => someBuiltin mat
  };
<several tests involving non-empty matrices here>
```

instead of

```ocaml
let myProc: matrix('a) => matrix('a) = mat => someBuiltin 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 (11 points)

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

6 Well-Ordering Proofs (8 points)

Suppose that \(H\) is a function on the positive natural numbers that satisfies the following recurrence:

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

Farmer Spike’s sheepdog 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.