Transposing a Matrix

By a "matrix" here we mean a list of lists, where:
- Each inner list has the same length.
- So we are restricting to a 2-D matrix.

Example:
- \[
\begin{bmatrix}
[1, & 2, & 3, & 4, \\
5, & 6, & 7, & 8, \\
9, & 10, & 11, & 12]
\end{bmatrix}
\]

Note: A matrix, even in higher dimensions, is a special case of an unlabeled ordered tree.

How to Transpose Functionally

- The firsts of each row become the first row of the transpose.
- \[\text{transpose}(M) = \text{map}(\text{first}, M) | \ldots???\ldots\]
- Now try to get recursion to do the work for us.
- \(M\) without the first column is \(\text{map}(\text{rest}, M)\).
- Transposing that gives us the rest of the rows.

Rule Summary

- \(\text{transpose}(\text{[] } | \ldots) => \text{[]}\);
- \(\text{transpose}(M) => [\text{map}(\text{first}, M) | \text{transpose}(\text{map}(\text{rest}, M))];\)
- For completeness, we may wish to add as a first rule:
  - \(\text{transpose}(\text{[]} ) => \text{[]}\);

Check by Rewriting

- \(\text{transpose}([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]))\)
- \([\text{map}(\text{first}, \ldots) | \text{transpose}(\text{map}(\text{rest}, \ldots))];\)
- \(\text{transpose}([[2, 3, 4], [6, 7, 8], [10, 12, 13]];))\)
- \(\text{transpose}([[3, 4, [7, 8], [11, 12]];])\)
- \(\text{transpose}([[2, 3, 4], [6, 7, 8], [10, 12, 13]];))\)
- \(\text{transpose}([[4, 8, 12], [5, 6, 7], [9, 10, 11]]);\)
- \(\text{transpose}([[5, 9], [2, 6, 10]; [3, 7, 11]; [4, 8, 12]; [11, 12];])\)
- \(\text{transpose}([[1, 5, 9]; [2, 6, 10]; [3, 7, 11]; [4, 8, 12]; [11, 12];])\)
- \(\text{transpose}([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]];))\)
- \(\text{transpose}(\text{[]} | \ldots) => \text{[]}\);
- \(\text{transpose}([[\text{[]} | \ldots]) => \text{[]}\)

The argument to \(\text{transpose}\) must have at least one row, even if that row is empty.
Functions Returning Functions

Anonymous Functions Again

- (X) \Rightarrow 5^X
  is a perfectly valid, although anonymous, function
- Functions in Rex are "first-class citizens":
  - They can be passed as arguments.
  - They can be put into lists.
  - They can be returned as values.
- A function-value returning function:
  - scaleBy(F) = (X) \Rightarrow F^X;
  - Here scaleBy(F) returns a function, the function that scales its argument by F.

Use of scaleBy

- scaleBy(F) = (X) \Rightarrow F^X;
- map(scaleBy(5), [1, 2, 3]) \Rightarrow [5, 10, 15]
- So we could define
  - scale(F, L) = map(scaleBy(F), L):

mapster

- mapster(F) = (L) \Rightarrow map(F, L);
- So mapster is a function such that, for any argument function F, returns a function that will map F over a list.

Curried Functions (yum!)

- scaleBy and mapster are examples of "Curried" functions, functions that take their arguments in sequential "tiers", each returning a function until the last argument.
- Another way to write them in Rex is:
  - scaleBy(F)(X) = F^X;
  - mapster(F)(L) = map(F, L);
- This emphasizes that scaleBy(F) has a meaning without the (X).

Anonymous returning Anonymous

- Yet another way to write scaleBy and mapster in Rex is:
  - scaleBy = (F) \Rightarrow (X) \Rightarrow F^X;
  - mapster = (F) \Rightarrow (L) \Rightarrow map(F, L);
- This clearly shows that the names scaleBy and mapster are incidental; the right-hand sides of the equations can be used as values as they are.
Curried Functions in Math, Science, and Engineering

- In a math, science, or engineering text, you might see $f_k(x)$.
- Typical nomenclature is that $x$ is the "argument" and $k$ is a "parameter".
- Actually both $k$ and $x$ may be viewed as arguments to $f$. Typically $k$ is "more fixed".
- We could define this as $f(k)(x) = \ldots$
- Then we can use $f(k)$ as $f_k$ would be used.

A function that "Curries" another function

- Suppose that $f$ represents a typical two-argument function, i.e. $f(x, y)$ is meaningful.
- Define $\text{curry}(f)(x)(y) = f(x, y)$
- $\text{curry}(f)$ is a curried version of $f$.
- $\text{curry}(f)(x)$ is like "$f(x, \_)$" in engineering books.
- Example: $\text{scale}(K, L) = \text{map}(\text{curry}(*)(K), L)$

A Cool Example

- We know that an association list can be viewed as an implementation of a function (with a finite domain):
  $$\text{["Jan", 31], ["Feb", 28], ["Mar", 31], ["Apr", 30]}$$
- But we cannot simply apply this list as a function:
  $$\text{["Jan", 31], ["Feb", 28], ["Mar", 31], ["Apr", 30]}(\text{"Feb"}) \Rightarrow 28$$
  not in `rex`, at least.
- We have to use the assoc function instead:
  $$\text{assoc}(\text{"Feb"}, \ldots \text{list above} \ldots ) = \text{["Feb", 28]}$$

Making Fun of Association Lists

- makeFun(Alist) returns a function that can be applied as an ordinary function:
  $$f = \text{makeFun}([\text{"Jan", 31}, \text{["Feb", 28], ["Mar", 31], ["Apr", 30]}]);$$
  $$f(\text{"Feb"}) = 28$$
- To define makeFun, we use the idea of a function returning a function:
  $$\text{makeFun}(\text{Alist}) = (\text{Arg}) \Rightarrow \text{second}(\text{assoc}(\text{Arg}, \text{Alist}))$$
- Using makeFun "captures" the Alist in the resulting function.

Composing Functions

- The composition of two functions, $f: T \to U$
  $g: S \to T$
  is the function, call it $h$ for now, such that
  $h: S \to U$
  and for every $x$ in $S$, $h(x) = f(g(x))$.
- The composition is sometimes written using an operator $\circ$:
  $f \circ g$
- If anonymous functions are supported, we don't need the "call it $h$" aspect:
  $$\text{compose}(f, g)(x) = f(g(x))$$
  or, equivalently,
  $$\text{compose}(f, g) = (x) \Rightarrow f(g(x))$$
Composing can make things more efficient in some cases

- \( \text{map}(f, \text{map}(g, L)) = \)

Searching Graphs and Trees

Searching a Graph

- Suppose we want to find whether there is a node with a certain property \( P \) reachable from a node in a graph.
- Rather than assume a specific representation such as an arc-list, use abstraction:
  - Assume we have a function targets such that targets(Graph, Node) is the list of targets of the node.

Finding a node in a graph

- Let findNode be the function to be computed. (call findNode(Graph, Node, P), where Node is the starting node, \( P \) is a predicate on nodes.)
- Success indicator convention:
  - If a node is found, return a list of just that node, \( [\text{Node}] \).
  - If no node is found, return the empty list, \( [\ ] \).

Finding a node using recursion

- \( \text{findNode}(\text{Graph}, \text{Node}, P) \Rightarrow P(\text{Node}) ? [\text{Node}] ; \)
- \( \text{findNode}(\text{Graph}, \text{Node}, P) \Rightarrow \)
  - \( \text{findInTarget}(\text{targets}(\text{Graph}, \text{Node}), \text{Graph}, P) ; \)
  - i.e. find one among the targets of Node

Finding a node among targets

- \( \text{findInTargets}([], \text{Graph}, P) \Rightarrow [\ ] ; \)
- \( \text{findInTargets}([\text{Target} | \text{Targets}], \text{Graph}, P) \Rightarrow \)
  - found = \( \text{findNode}(\text{Graph}, \text{Target}, P) ; \)
  - \( \text{found} \in [\ ] ? \text{found} ; \text{findInTargets}([\text{Targets}], \text{Graph}, P) \)}
  - returned value
  - local definition
  - recurse
The relationship between find and findInTargets is that of mutual recursion.

- find delegates work to findInTargets
- findInTargets delegates work to find
- This approach seems natural in this problem.

The previous solution will work if the graph is acyclic.

- If not acyclic, it may work in some cases, and loop infinitely in others.
- So it doesn't really "work".
- How can we fix this?

Since the graph is finite, infinite looping can only occur when the same node recurs on a path.

- By keeping track of the nodes on the path from the starting point, we can check whether a node recurs.

The version of find presented previously is called depth-first search.

- The other prevalent form of find is breadth-first search.

We include Target in the list above so that we never do a recursive find from a node more than once.

However, we may still search from the same node more than once. Why? Is this preventable?
Comparative Strengths

- Breadth-first has the advantage of finding the **shortest** path to a desired node.
- Depth-first is easier to implement and is more space-efficient.

Underlying Data Structures

- Depth first uses recursion; the underlying structure is a "stack".
- Breadth-first uses a "queue" (first-in-first-out list).
- More on this when we discuss implementation of information structures.
- See also the discussion in text, sec. 4.15.

Infinite Lists

- In Rex, lists can be conceptually infinite.
- Infinite lists allow us to model "processes" that continue processing forever; a list is used as a data stream from one process to another.
- The lists don't really take up infinite space; they are computed incrementally as needed.

Built-in infinite list functions

- `from(0)` → `[0, 1, 2, 3, 4, ...]`
- `map(sq, from(0))` → `[0, 1, 4, 9, 16, ...]`
- `map(*, from(0), from(0))` → `[0, 1, 4, 9, ...]`
- `primes()` → `[2, 3, 5, 7, 11, ...]`

Not all functions make sense on infinite lists

- `append(from(0), range(1, 10))` → ?? (be ready with control-C)
- `append(range(1, 10), from(11))` is ok
- `reverse(from(0))` → ?? (uh-oh)
- `range(0, Infinity)` should work

How it's done: smoke & mirrors

- Delay operator: `$`
  - `$expression...` means "defer computing the value of expression until needed".
- Application:
  - `f(g(X), $h(Y))`
  - where `h(Y)` is some really expensive computation that might not be needed.
Infinite lists == $ ?

- from(N) = [N | $ from(N+1)];
  
  from(0) => [0 | $ from(1)]
  => [0 | [1 | $ from(2)]]
  => [0 | [1 | [2 | $ from(3)]]]
  => [0 | [1 | [2 | [3 | $ from(4)]]]]
  == [0, 1, 2, 3, 4, ...]

A good challenge

- Define pairs so that

  pairs(from(0), from(0)) =>
  [ [0, 0],
    [0, 1], [1, 0],
    [0, 2], [1, 1], [2, 0], ... ]

  Every pair must be included eventually.

- A "proof by programming" of a mathematical truth.