Solve.hs

Module 1 Exercises - Lists and Loop Patterns

Lecture 1 - Introduction and Pattern Matching

Haskell Installation and Setup

To start, make sure you've downloaded and unzipped the `Solve-hs.zip` file from the lecture! The link for it should be right under where you got this exercises document. This zip directory contains a working Haskell project that you'll use for the rest of the course.

To run the code and tests, you will, of course, need to make sure you've installed the Haskell toolchain on your machine (if you haven't already). The easiest way to do this is with GHCup. You can find the full instructions online here: https://www.haskell.org/ghcup/install/

For a detailed breakdown of setting up your Haskell toolchain, you can also supplement this course with our free course **Setup.hs**: https://academy.mondaymorninghaskell.com/p/setup-hs

Install Test Executable

When your Haskell toolchain is working, change directory into our project root and build the project.

```
$ cd Solve-hs
$ stack build
```

This could take a few minutes, especially if you need to install a new GHC version.

Next, run `stack install`, which will install the `shs-test` command.

```
$ stack install
```

This will allow you to easily run the test commands for each lecture. For example, you'll run this command for this lecture's tests (Module 1, Lecture 1):

```
$ shs-test 1-1
```

Defining Our List

Now let's write some simple code to complete these exercises. Open the `MyList.hs` file in the `src` directory (all our code modules are in this directory).

Fill in the definition for `MyList` that we went over in the lecture. Then implement the four basic functions in the first section of the file: `null'`, `head'`, `tail'`, and `cons'`. These will use basic pattern matching on their function arguments. Note that `head'` and `tail'` can throw an `error` for empty lists.

Practice Problems

Now complete a few basic problems to practice pattern matching some more.

Sum 3

Start off in `src/M1Lecture1.hs`. Fill in the `sum3` function. This should return the sum of the first three elements in a list of integers. If there are fewer than 3 elements, just take the sum of all elements present.

Coordinate and Direction Utilities

Now go into `Utils.hs`. There are a few types in this file that will be referenced throughout the course, since many problems can be described using 2D space. Fill in a few helper functions on some of these types:

`stepD4` - Given a 2D coordinate and a direction, take "one step" in the given direction and return the new coordinate.
`moveD4` - Similar to `stepD4`, but now you can move more than one space (this is the first parameter).
`moveD4'` - Same as above, but this wraps the direction and distance of the movement in a new data type, which is occasionally more convenient.
`stepD4f` - These final two functions use `Coord2f`, which has decimal values for the coordinates.
`moveD4f` - Same as stepping, but with a custom distance.

Note that for all of these, the tuple is an X/Y coordinate pair, meaning that moving "right" increases the X-coordinate, which is the first element of the tuple. Moving "up" increases the Y-coordinate, which is the second element.

First Travel

Now go back to `M1Lecture1.hs` and fill in the function `firstTravel`. Given a list of 2D coordinates, return the Manhattan distance (absolute x-difference + absolute y-difference) between the first two coordinates. If there are fewer than 2 coordinates, return 0.

Run the test command (`shs-test 1-1`) and ensure all the tests pass. Then you can move on to the next lecture.

Lecture 2 - Basic Recursion

List Functions

To start, implement a series of simple functions in `MyList.hs` for our custom type:

`atIndex` -- Return n-th element of a list, equivalent to the `(!!)` operator
`last'` -- Return the last element of a list
`elem'` -- Return a boolean for whether or not the item is in the list.
`find'` -- If the given element is in the list, return it as `Just`. Otherwise return `Nothing`.
`and'` -- Return `True` unless at least one of the elements of a boolean list is `False`.
`or'` -- Return `False` unless at least one of the elements of a boolean list is `True`.
`all'` -- Given a predicate, return `True` unless one of the elements evaluates as `False`.
`any'` -- Given a predicate, return `False` unless one of the elements evaluates as `True`.
`isPrefixOf'` -- Determines if the first list is a "prefix" of the second.
`isInfixOf'` -- Determines if the first list can be found anywhere within the second.

Each of these requires simple recursion, without needing to accumulate any value.

Practice Problems

Now solve the practice problems in `M1Lecture2.hs`.

Parallel Cars

Imagine two cars are driving in adjacent lanes down a street. Each car has an initial position (`x1` and `x2`), and a velocity (`v1` and `v2`), and these inputs are given as a tuple for each car `(x1, v1), (x2, v2)`. With each timestep, each car will move down the street based on its velocity.

The `parallelCars` function should return a boolean value telling us if these cars will ever have the same `x` location and the *end* of a timestep. A few examples:

`(1, 3) (5, 1)` - This should return `True`. After the second timestep, both cars will be at 7.
`(5, 4) (7, 5)` - This should return `False`. The second car starts farther along and is going faster, so the first car will never catch up.
`(2, 4) (9, 1)` - This should return `False`. After the third step, the first car will pass the second car, reaching `x=14` while the second is at `x=12`, and the second will never catch up.

All values are given using the `Word` type (Haskell's unsigned integer), so they can never be negative. Cars only move "to the right".

Finding the Key

In `findKey`, you have two arguments: a *key* string, and a *search* string. You should return `True` if all of the letters in the key string can be found in the search string. Order does not matter, but case does. Only one instance of a key-letter needs to be in the search string. So `findKey "aa" "a"` should be `True`.

First Proper Factor

In `firstProperFactor`, you're given a list of integers. Return the first element of the list that is a proper factor of the final element in the list. That is, the number should evenly divide the final number, and it should be strictly smaller. So `[5, 7, 24, 8, 9, 12, 24]` should return `Just 8`, while `[9, 13, 24, 24]` should return `Nothing`.

Are All Factors?

There are two inputs to `areAllFactors`. The first is a list of indices, the second is the list of numbers we're interested in. For each index, you need to determine if the value in the numbers array at that index evenly divides the **sum** of the numbers list. If all the indices pass this test, return `True`. Here are a couple examples:

```
areAllFactors [0, 2] [2, 1, 3]
areAllFactors [1, 3, 5] [10, 12, 14, 16, 18, 26]
```

The first should return `True`. The sum of the numbers list is 6. The indices 0 and 2 point to 2 and 3, which are both factors of 6.

The second example should return `False`. The sum of these numbers is 96. The indices refer to the numbers 12, 16, and 26. While 12 and 16 are factors of 96, **26** is not.

Do not worry about checking for out-of-bounds indices.

Has Incrementing List

Once again, your function takes two integer list parameters. This time, we're trying to find if any of the numbers in the first list is the start of an incrementing sequence that is at least 3 long in the second list. For example, if the following list is our second input:

```
[5, 1, 2, 3, 17, 43, 18, 19, 20, 21, 42, 43]
```

There are three values that start incrementing sublists for this list: 1, 18, and 19. If the first input contains any of those values, the function would return `True`, otherwise `False`.

Hash Until Increment

The `hashUntilIncrement` function takes a `ByteString` key and an initial integer value. At each step of iteration, you'll concatenate the integer to the key and produce its MD5 hash. Your job is to find the smallest number, starting from the initial value, for which the resulting has begins with 3 bytes that represent consecutive numbers. You can use the `mkHash` function which will produce your hash, and the `first3` function, which will give you the first 3 bytes as integer values IF they represent numeric bytes (values 48-57 on the ASCII chart).

Test your answers by running:

```
$ sh-test 1-2
```

Lecture 3 - Recursion with Accumulation

List Functions

Implement a few more functions in `MyList.hs`. These functions require the accumulation of a value in a recursive function.

`length'` -- Return the length of the list
`sum'` -- Return the sum of a list of integers
`product'` -- Return the product of a list of integers
`maximum'` -- Given a list of orderable elements, return the largest (error on empty list)
`minimum'` -- Given a list of orderable elements, return the smallest (error on empty list)
`elemIndex'` -- Given an element, return the index of that element in the list (or `Nothing`)

Practice Problems

Now let's try some practice problems in `M1Lecture3.hs`.

Elevation

Your input for the function `elevation` is a string composed of the letters `u` and `d`. Each of these represents a command to either shift an airplane's elevation up (`'u'`) or down (`'d'`) by 100 feet from its starting altitude. Your answer should be the net elevation change.

So for example, if your input is the string `"udduu"`, the airplane moves up 100 feet, down 200 feet, and then up 200 feet for a final answer of 100.

Make Digits

For the `makeDigits` function, you're given an integer. Return a list of the digits in that number, starting with the least significant (1's place). So if your input is 3042, you should return `[2, 4, 0, 3]`. If your input is 0, you can return an empty list.

Build Number

Now fill in `buildNumber`, which reverses the process, except that the input list of digits starts with the *most* significant digit. So if your input is `[2, 4, 0, 3]`, you should return 2403. You'll want to use the `length` function to get the correct power of 10.

Viral on X

You're trying to go viral with a new post on X (formerly known as Twitter). The function `viralCount` takes two parameters. The first is the number of people you are sharing it with, and then the second parameter is the number of hours you are measuring. Each hour, one third (rounded down) of the people seeing your tweet for the first time like the tweet and then share it with four of their friends. The new friends then see it at the start of the next hour. How many people like your tweet at the given number of hours?

For example, `viralCount 13 3` should return 15.

In the first hour, **4** people like and share the tweet, meaning that they share it with 16 new people. These people see it at the start of the second hour, and **5** of them like it, sharing with 20 people. Then **6** of these people share the tweets during the third hour. This is the final hour we care about, so the total is 4 + 5 + 6 = 15.

Add Quads

Given two lists of integers, we want to add the product of "quads" formed by the two lists. The first quad consists of the first two numbers in each list. The second quad has each of the second two numbers. Once you run out of numbers in one or both lists, treat all other values in the quad as 1.

So for example, suppose we have these lists:

```
addQuads [1, 6, 5, 3, 2] [8, 3, 2, -3, 4, 7, 9]
```

The first quad is `(1 * 6 * 8 * 3)`. The second quad is `(5 * 3 * 2 * -3)`. The third quad is `(2 * 4 * 7)`. The final quad is just 9. We add these together to get 144 - 90 + 56 + 9 = 119.

Count Characters

In `countChars`, you're given an input string and you're supposed to find the number of characters represented by the string. Why can't you just use length here? The catch is that there are instances where we use multiple characters to represent a single character.

For example, you have basic escaped characters like quotation marks (`\"`), tabs (`\t`) and new line characters (`\n`), which have a backslash in front of them. These take up two characters, but should only count for 1.

Then some characters are represented in octal. These consist of 3 characters preceded by the prefix `\o`, like `\o231`. This means that 5 characters in the input string only represent a single character.

Finally, there are hexadecimal characters. These are similar to octal, but the prefix is `\x` and there are two characters, like `\xf9`.

Using these rules, count the number of characters your input string actually represents.

Test your answers by running:

```
$ sh-test 1-3
```

Lecture 4 - Tail Recursion

List Functions

Search for the "Tail Recursion" section of the `MyList` module. In this section, you'll essentially re-implement several of the functions you wrote in the last couple lectures, only this time you'll use tail recursion.

Practice Problems

Your practice problems are in `M1Lecture4.hs`.

Elevation Redux

Recall our function elevation from the last lecture. For the function `elevationRedux`, you're processing the same input. Only this time, you're solving a different problem. It turns out the airplane is flying at its minimum safe altitude at the start. So you need to find the first index, if any, where the input commands will take it to a negative net altitude. If you reach the end and it has never gone negative, you should return `Nothing`. Treat the initial value as index 1, not index 0.

So, if your input is the string `"udduu"`, the airplane will have a net altitude of 100 after the first command, 0 after the second, and then -100 after the third command. So you should return `Just 3`. If the input is `"duu"`, you would return `Just 1`, since the first command takes it below its safe altitude.

Build Number Better

In lecture 2, we had `buildNumber`, where your job was to construct a number from its digits. However, using only basic recursion, you needed to recalculate the length of the remaining digits each time, which is inefficient. Write `buildNumberBetter`, which should do the same task, but uses tail recursion to avoid this recalculation.

Maxes and Mins

Fill in the implementation for `maxesAndMins`. Given a sequence of integers, return a 2-tuple with the number of times the series encounters a new maximum and a new minimum. The first value sets a baseline and does not count either as a max or a min. If the input list is empty, you should return `(0, 0)`.

For example, given the series `[1, 5, 5, 3, 2, 0, 6, -1, -1, 10]`, you should return `(3, 2)`. The new maximums are 5, 6, and then 10. The minimums are 0 and -1.

Fireworks

It's New Year's Eve, and your pesky neighbors are setting off fireworks. You need to figure out how many are going to land on your house!

Imagine that your house is situated on a 1-dimensional axis, with two neighbors, one to the left (`neighbor1`) and one to the right (`neighbor2`) of your house. They're each setting off fireworks from a particular point (representing the location of their house), with each firework having an offset that is either positive, meaning it lands to the right of its starting location, or negative, meaning it lands to the left. The `fireworks` function takes several parameters to represent these:

`(h1, h2)` is a tuple with two values representing the interval covered by your house.
`n1` is the location of `neighbor1`, who will always be to the left of your house.
`n2` is the location of `neighbor2`, always to your right.
`f1s` are the offsets for the fireworks from `neighbor1`
`f2s` are the offsets for the fireworks from `neighbor2`

Determine how many of their fireworks will land within the (inclusive) interval of your house. For example, suppose we have:

```
fireworks (5, 10) 2 14 [9, -3, 8, 1, -1, 4, 3] [-5, 1, 2]
```

We should get a total of 4. From the left, the `8`, `4`, and `3` fireworks hit our house. The `9` overshoots us, and the rest fall harmlessly to our left. From the right, only the `-5` hits us.

Math 2D

Define a function that will loop through a 2D matrix (list of integer lists) and accumulate a result value using the following process:

1. Start with 0
2. For each even-index column in each row (starting from 0-index), add the matrix value to your accumulated value.
3. For each odd column, multiply the accumulated value by the matrix value
4. Repeat for each row.

So for example, if we have a 2x3 matrix like so:

```
4 3 -2
6 2 13
```

The accumulated value after the first row should be `((0 + 4) * 3) - 2`, giving 10. Then we feed 10 into the second row to get `((10 + 6) * 2) + 13`, giving a final result of 45.

Your code should have essentially the same effect as the following Python code:

```
def math2d(values):
  result = 0
  for i in range(0, len(values)):
    for j in range(0, len(values[0])):
      if j % 2 == 0:
        result += values[i][j]
      else:
        result *= values[i][j]
  return result
```

Triplet Sums

Given a list of integers, add them up in this particular way. For each three numbers you encounter, you should add the first two, and then take the modulo with the third number. Add up all these results to get your final answer. Note our triplet division is different from the largestTriplet function you've seen in the lecture slides. You should essentially pre-divide the list into disjoint groups of 3, rather than creating overlapping triplets. If the length of the list is not divisible by 3, just add the remaining numbers. So with this example:

```
tripletSums [6, 7, 4, 2, 35, 17, 3, 4]
```

The answer should be 11, because we have `((6 + 7) `mod` 4) + ((2 + 35) `mod` 17) + (3 + 4)`, which is 1 + 3 + 7.

Test your answers by running:

```
$ sh-test 1-4
```

Lecture 5 - List Accumulation

List Functions

In the next section of the `MyList` module, you'll implement several functions where you now have to accumulate a whole list of values. You'll want to use tail recursion here. You'll also implement `reverse'` first, since you can then use it to help with the other functions!

`reverse'` -- Reverse the input list
`append'` -- Combine two lists
`findIndices'` -- Return a list of indices of the list where the element satisfies the predicate
`isSuffixOf'` -- Return true if the first list is a suffix of the second (hint: use other functions!)
`map'` -- Return a list where each element of the input list has been passed through the function
`filter'` -- Return the subset of the input list that satisfies the predicate
`snoc'` -- Append an element to the end of a list ('snoc' is the reverse of 'cons')
`init'` -- Return all elements of a list except the last (error on an empty list)
`concat'` -- Combine a list of lists into a single list
`concatMap'` -- Map a function that produces lists, and concat the lists together
`zip'` -- Combine two lists (of potentially different types) into one list of tuples that is as long as the shortest input. Drop items from the longer list that are beyond the size of the shorter list.

Practice Problems

Enumerate

In Python, you can use the `enumerate` function with an iterable collection. This is often used in loops when you want the index as well as the object accessible:

```
for (i, item) in enumerate(my_list):
  print(f"Index is {i} item is {str(item)}")
```

For your first practice problem, implement this function in Haskell! Given a list of items, return a list of tuples, where each tuple pairs the object with its integer index, starting at 0. If you have these inputs:

```
enumerate ["Hello", "World", "!"]
```

Your output should be `[(0, "Hello"), (1, "World"), (2, "!")]`.

Calculate Velocities

Imagine an object moving along a one-dimensional axis. Your input is a time-series of its position. That is, it is a list of tuples where each tuple tells its position along the axis and the time at which that position was measured.

Your job is to return the list of average velocities between each point. Velocity is calculated by the change in position divided by the change in time. So with this input:

```
calculateVelocities [(1.0, 0.0), (2.0, 1.0), (5.0, 3.0), (4.0, 3.5)]
```

You should return `[1.0, 1.5, -2.0]`. Your output should always be one item shorter than the input. You can assume that the time values are strictly increasing in the input list.

Palindromes

Given a string, write a function that returns `True` if the string is a palindrome (same forwards as backwards) *after* having removed spaces. So strings like `"abba"` and `"a man a plan a canal panama"` should return True. All input letters will be lowercase.

Make Digits Redux

Rewrite your `makeDigits` function from the last lecture, except this time it should return the digits starting with the *most* significant. So with the input of 3042, your result should be `[3, 4, 0, 2]`.

All Factors

Given a list of integers, return a list of all numbers that are a factor of at least one of the inputs. Your output list should be in ascending order with no duplicates. For example, if the input is `[8, 4, 9]`, your output should be `[1, 2, 3, 4, 8, 9]`.

Package Arrival

You're working on constructing a bike shed, but unfortunately the shed you're ordering is arriving in several different packages at different times. The bad news is that you have to get started by a certain time in order to complete the project today, otherwise it isn't worth even starting. The good news is that you don't need *all* the packages to get started.

So with this in mind, fill in the function `canStartShed`. The function parameters are the time you need to start assembling the shed by, the number of packages you need to get started, and the arrival times of each package. So for example:

`canStartShed 14 3 [12, 9, 15, 18, 20, 5]` should return `True`. By the required starting time (14), 3 packages will have arrived (5, 9, and 12), and this is the number you need to get started.

However, `canStartShed 10 4 [12, 9, 15, 18, 20, 5]` should return `False`. Only two packages will have arrived by your required starting time (5, 9), and you require 4.

Test your answers by running:

```
$ sh-test 1-5
```

Lecture 6 - Folding

List Problems

In the next section of the `MyList` module, you'll find a few different functions related to folding. First, you'll implement `foldl''` (basic, non-strict left-fold), and `foldr'`, the core folding functions.

Next, you'll implement `scanl'`. A scan is like a fold, except that instead of returning a single result value at the end, you return a whole list of values! This list should include each accumulated stateful value. The scan result should include the first AND last values. So its size should be 1 greater than the size of the input list!

For example, `scanl' (/) 64.0 [4, 2, 4] = [64.0, 16.0, 8.0, 2.0]`

Finally, you'll implement two more functions from before, except this time implement them using folds instead of raw recursion! These are `sum'''` (this is the last `sum` implementation, I promise!) and `map''`.

Practice Problems

Volume and Surface Area

For the function `volumeSurfaceArea`, your input is a list of integer 3-tuples representing the dimensions (length/width/height) of boxes. You should return two values - the sum of the volumes of these boxes and the sum of their surface areas.

Count Most Adjacent

Your input to `countMostAdjacent` is a list of integer pairs. In each pair, the first number is the "value" and the second number is the number of times that value appears in a set. You should return the largest count of values you can get by selecting two adjacent values.

For example, with `[(1, 4), (2, 3), (3, 5), (4, 2)]`, you should return 8, since we can combine the 2's and 3's. However, with `[(1, 4), (2, 3), (4, 5), (5, 2)]`, you should return 7. The middle values (2 and 4) are no longer adjacent, so the best adjacent pairing is 4 and 5.

Robot Hits

Imagine you have two robots on a number line, robot X and robot Y, and their job is to play "whack-a-mole" and knock out spikes that appear on the line. Each time a spike appears, the robots move toward the spike at the same rate until one of them hits it. If they both reach the spike at the same time, they both hit the spike (it is a tie), and then (only in this case), robot X moves two spaces to the left, and robot Y moves two spaces to the right. If it is not a tie, there is no movement of either robot after hitting the spike.

The inputs to `robotHits` are the initial position of robot X, the initial position of robot Y, and a list of spike locations. You should return a 3-tuple with the number of times robot X gets the hit (excluding ties), the number of times robot Y gets the hit (excluding ties) and the number of ties.

Math 2D

Reimplement `math2D` from lecture 4, only now you should make use of folds!

Seam Carving

"Seam carving" is an algorithm that allows you to trim an image in one dimension (we'll assume vertically) by removing the least important vertical "seams" in the image. This gives us a narrower image that retains the most important features of the original.

The function input is a nested list of integers. Each sub-list represents a set of "energy" values for each pixel in one row of the image. (Each row will have the same number of values; do not worry about validating that). Your job is to find the lowest cumulative energy path from the top row to the bottom row, where the path takes one element from each row. You only need to return the final amount of energy, not the path itself.

To do this, keep track of a list of the minimum total energy you've seen for each column. For the first row, this just means copying the existing energy values. For each subsequent row, you'll add the value in each column to the minimum value taken from the three values above that column from the previous "minimum energy row" (directly above, above and to the left, or above and to the right).

After processing the final row of the input, you should be able to take the minimum value and return it.

Note: This is our first example of a "dynamic programming" algorithm. We'll discuss the paradigm more in Module 3. You'll also get to redo this particular algorithm more efficiently using other data structures that we'll discuss in Module 2 (so don't worry about inefficiencies caused by list operations like indexing). You'll also learn how to return the full seam path itself.

Manhattan Travel 2D

Imagine we have a path composed of 2D coordinates. We plan to travel this path in a very specific way, using only the 4 cardinal directions (Up/Down/Left/Right). Whenever we go from one point to another, we always cover the *vertical* distance first, and *then* the horizontal distance.

So if our path is `[(0, 0), (3, 4), (6, 2)]`, our path must look like this:

```
.......
.......
>>>2...
^..v...
^..v>>3
^......
1......
```

We go up from the first point until we're at `(0, 4)`, then right to the second point. Then we go down to `(3, 2)`, and finally over to the third point.

Suppose you're given a path and a **fractional value** (between `0.0` and `1.0` inclusive). The fraction tells you what percentage of the path you want to travel (by distance). Your job is to return the final coordinate you end at.

So in the above example, a fraction of `0.0` tells you to stay at `(0,0)`. A fraction of `1.0` tells you to travel all the way to `(6,2)`. Suppose we get the fraction `0.5`. Our path's total length is 12, so we want to travel a total distance of 6. Following the path we described, this would leave us at the position `(2,4)`. If your input is an empty list, you can just return `(0,0)`.

Test your answers by running:

```
$ sh-test 1-6
```

Lecture 7 - Higher Order Function Patterns

List Functions

To begin, implement a few functions incorporating the `By` and `On` patterns.

`maximumBy'` -- The largest element in the list based on the comparator function
`sort'` -- Sort the input list
`sortBy'` -- Sort the input list based on a comparator
`sortOn'` -- Sort the input list based on an evaluator function
`group'` -- Return a list of lists, grouping adjacent equivalent elements
`groupBy'` -- Same as group, but using a custom equality predicate

When it comes to sorting, you can stick to the "basic" quicksort pattern we went over in lecture. Remember that you can write functions in terms of each other!

Practice Problems

Student Awards

The `Student` type records the name of a student as well as their scores in various subjects. Your job in the `studentAwards` function is to return 3 names. The first should be the student with the highest score in math, the second should be the highest in science, and the third should be the highest in literature. If the list is empty, you should return 3 empty strings.

Any Overlap

For the function `anyOverlap`, you're given a list of integer tuples. Your job is to determine if any of these tuples overlap with one another. Treat the intervals as inclusive on both sides, so that `(10, 12)` and `(12, 15)` are considered overlapping. (This can be done in `O(n log n)` time, where `n` is the number of intervals).

Build Intervals

Your input is a list of boolean values. Each one specifies if the corresponding integer point on a number line is part of an interval (like we had above) or not. The second function parameter indicates the starting index of the list. Your job is to turn this list of booleans into a list of disjoint intervals. So for this example:

```
buildIntervals 2 [True, True, False, True, False, True, True, True]
```

You should return a list of three intervals: `[(2, 3), (5, 5), (7, 9)]`

The first two `True` values form the interval `(2, 3)`, because 2 is our starting value. Then index 4 is `False`. The next interval just has size 1. Then after the `False` at index 6, we have three `True` values forming the interval `(7, 9)`.

Anagrams

Given a list of words, return a list of string-lists where each list contains all the words from the original input that are anagrams of each other. For example, suppose our input is:

```
["cab", "box", "ox", "abc", "xo", "oxb"]
```

You should return:

```
[["abc", "cab"], ["box", "oxb"], ["ox", "xo"]]
```

Each individual list should be sorted, and you should sort the lists based on their first element.

Build a Maze

In `buildMaze`, you're given two parameters. The first is a string representing an encoded 2D maze. It consists of the characters `'.'` and `'x'`. The period characters represent empty space in the maze, and the `'x'` characters are walls. You're also given an integer for the number of rows this maze is supposed to have, which is guaranteed to be a divisor of the length of the input string.

Your job is to return a nested association list, with the type `[[((Int, Int), Bool)]]`. Each item is a cell in the maze, with its row, column, and a bool indicating if the space is a wall. Each inner list denotes one row of cells For example, given the following inputs:

```
buildMaze "..x...x.." 3
```

You should return this result:

```
[ [((0,0), False), ((0,1), False), ((0,2), True)]
, [((1,0), False), ((1,1), False), ((1,2), False)]
, [((2,0), True), ((2,1), False), ((2,2), True)]
]
```

Incrementing Chunks

Given a list of integers, divide it into lists that are monotonically increasing (each value is 1 more than the last). For example:

```
incrementingChunks [1, 2, 5, 2, 4, 5, 6, 7, 10, 11]
```

Should return this result:

```
[[1, 2], [5], [2], [4, 5, 6, 7], [10, 11]]
```

Test your answers by running:

```
$ sh-test 1-7
```

Lecture 8 - Set Operations

List Functions

In the `MyList` module implement a few set-based list functions. Don't use the `Set` type for these, just rely on the `MyList` type, even though it's less efficient. We'll learn about the `Set` type (and compare performance) in Module 2.

`nub'` -- Return the input list, with all duplicates deleted
`delete'` -- Delete the first occurrence of the given element in your list
`intersect'` -- Return a new list containing all elements present in both lists
`union'` - Return a new list containing all elements present in either list.

Note that for `intersect'` and `union'`, if the first list contains duplicates, the resulting list should as well. But if the second list contains duplicates, there should be no repetition of these elements. When taking the union, items from the second list that are also present in the first list should not appear again.

Practice Problems

Common Token

Your input is a list of strings, and you are guaranteed that the length of the list is divisible by 3. Divide the list into groups of 3. Within each group of 3, find the one character that is common to all 3 strings. Return a string composed of these characters. All characters will be lowercase. For example, if your inputs are:

```
input
reject
tornado
realize
criminal
isotope
left
egregious
rife
```

Your output should be `"tie"`, since `'t'` is the only letter common to the first 3 words, then `'i'` is the only common letter in the second 3, and `'e'` is the only common letter among the final 3. You should use `error` in any cases where you can't find a proper solution, like if the input list isn't divisible by 3, or if you cannot find a common token, or if there are multiple common tokens.

Compartmentalize

Your input is a single string, as well as a number. It is guaranteed that the length of the string is divisible by the number. Your first job is to split the word into the number of parts given by the number; these are your "compartments". Then you must find the letter that is common to each compartment.

So if your inputs are `"ibegpgdeniemquem"` and `4`, you should return `Just 'e'`. The four compartments are `'ibeg'`, `'pgde'`, `'niem'` and `'quem'`, and `'e'` is the only common letter here. If there is no common letter or multiple common letters, you should return `Nothing`.

Traverse 2D

Your inputs are a starting 2D coordinate and a list of directions (Up/Down/Left/Right). Your job is to follow the directions you're given, starting from the given coordinate, and keep track of the coordinates that are visited. You should return the number of unique coordinates that are traveled.

So with the following function call:

```
traverse2D (1,1) [Up,Right,Down,Left,Up,Left,Down,Right,Up,Up]
```

Your path will look like this:

```
[(1,1),(1,2),(2,2),(2,1),(1,1),(1,2),(0,2),(0,1),(1,1),(1,2),(1,3)]
```

The path length is 11, but `(1,1)` and `(1,2)` are visited multiple times. Only 7 unique coordinates are visited, so you should return `7`.

Smallest Multiples

There are two input parameters to `smallestMultiples`, and both are integer lists. The first is the "search" list and the second is the "modulo" list. For each item in the modulo list, you'll select the smallest item in the search list that is a multiple of that number and save it in a return list. You should then remove that number from the search list. If no numbers are divisible, then you should not return any value for that modulus. For example, suppose we have:

```
smallestMultiples [16, 9, 12, 34, 26, 18] [4, 17, 3, 5, 6]
```

This should return `[12, 34, 9, 18]`.

Grabbing Letters

There are two inputs to the `grabbingLetters` function. The first is a list of strings. You should think of each string only as a collection of lowercase letters. You'll grab 3 of these collections and return the (sorted) set of unique letters you've collected from those three sets.

In this first version of the problem, you're not the one doing the picking. You've enlisted the help of three school children to help you. They each pick a stack and report the "index" they picked, and these 3 values (in a tuple) are what you get as your second parameter. They aren't programmers, so 1 refers to the first collection in the list.

The second two also forgot to account for the picks before them. Their indices are given based on the *remaining* collections. For example, suppose we have these inputs:

```
grabbingLetters ["ab", "ca", "bc", "ehis", "zafg"] (1, 2, 3)
```

Your output should be `"abcfgz"`. The first child picks the first collection `"ab"`. The second child picks the second of the *remaining* group, which is `"bc"`. The third child picks the third of the remaining three strings, `"zafg"`. After combining and sorting these letters, you get `"abcfgz"`.

Do not worry about validating the indices. There will be no out-of-bounds exceptions.

Grabbing the Most Letters

You've now decided that the kids made the problem a little more difficult, and you also generally get suboptimal results. So now you've decided you'll be the one picking the 3 strings, and you want to pick the 3 that give you the most letters. In the case of a tie for the most letters, you should pick the complete set of letters that comes first lexicographically. So using our earlier example:

```
grabbingLetters ["ab", "ca", "bc", "ehis", "zafg"]
```

The result should be `"abcefghisz"`.

Note: You can assume the scale of this problem is quite small. There will be no more than 10 strings, and each string is no longer than 10 letters.

Test your answers by running:

```
$ sh-test 1-8
```

Lecture 9 - Monadic Operations

List Functions

Implement the 3 basic monadic list functions discussed in the lecture:

```
mapM'
foldM'
sequence'
```

Practice Problems

To start, you'll re-implement a few functions from the last couple of lectures, except now you'll be implementing the functions monadically.

Common Token (with Fail)

Re-implement `commonToken` from the last lecture, but now using `MonadFail`. In the cases where you used `error` before, now use `fail`.

Compartmentalize (with Fail)

Similarly, re-implement `compartmentalize`, but use `MonadFail` *instead of* using `Maybe`. In the cases where you had previously returned `Nothing`, you should now use `fail`. (Incidentally, some of the test cases will actually still use `Maybe`).

Any Overlap (with Logging)

Now you'll re-do `anyOverlap` from a couple lectures ago, but this time, you'll use `MonadLogger`. If you didn't before, you should use an implementation that first sorts the intervals (by starting index) and then goes through each adjacent pair and evaluates if they overlap.

Each time you evaluate a pair of intervals, you should log a message with the result using this format:

```
(1,3) and (5,6) do not overlap!
(2,4) and (3,5) overlap!
```

If there are no overlaps, then you should also print one final message:

```
No overlap found!
```

The tests will validate that these messages are printed.

Traverse 2D (with Logging)

Re-implement the `traverse2D` function from last lecture, but now have it implement `MonadLogger` so that messages are logged. Whenever you visit a space (whether or not you've seen it before), you should now log a message like this:

```
Visiting: (1,1)
```

Note that you should not log a message for the initial location.

Queue Implementation

We've restricted ourselves to using lists throughout this whole module, and Haskell's list type acts best as a "stack" data structure - you can only efficiently add or remove items from one end (that is, "Last-In-First-Out" or LIFO).

A *Queue* is a list-like data structure that makes it efficient to add items to one end and then remove them from the other ("First-In-First-Out" or FIFO). In Module 2, we'll learn about the *Sequence*, which is Haskell's preferred queue data structure.

But even just with what you know about lists, you can implement a very basic queue! In the module, you'll see the data type `ListQueue`, which has two list fields. You can modify the field names, but do not add any more fields to the type.

Your job is to implement the `enqueue` and `dequeue` functions, which both use the `State` monad since they change the underlying structure. You should write them so that the *amortized* complexity of each operation is `O(1)`. This means that a single operation might actually be `O(n)` (where `n` is the existing size of the queue), but any series of `n` operations is still `O(n)`.

Mixing Monads

Re-implement `math2d` from lecture 6, but using 3(!) different monads. First, you should use `MonadFail` and return an error if the rows do not all have the same size. Second, whenever you complete the processing for a row, you should log a message like so, where the first row shows up with index 1:

```
After row 1, value is 10
```

Finally, the whole function should operate by holding the accumulated value within the `State` monad. This means you should source the original value from the monadic context instead of beginning with 0. You also need not return the value explicitly. Consumers should pull the value out of the monad as well.

Test your answers by running:

```
$ sh-test 1-9
```

Lecture 10 - State Evolution Patterns

To start, in `M1Lecture10.hs`, implement the base function patterns we talked about in the lecture.

```
evolveState
evolveUntil
runFSM
runPDA
```

Practice Problems

Tree Growth

A magical tree is growing. Each year, it doubles its height and then grows another 3 feet. The `treeGrowth` function takes two parameters: an initial height, and the number of years it will grow. Determine how tall it is at the end of that period.

Fish Population

For this problem, you'll be calculating the number of fish in a small lake that initially starts with 1000 fish. Your function takes two inputs: the number of years to calculate, and the number of fish that are caught each year by the local population for fishing. Each year, there is a natural increase of 10% (rounding down) in the population.

Calculate the number of fish at the end of the n-th year (based on the input). For each year, you can calculate the natural increase first, and THEN subtract the number of fish eaten.

Counting Cars

In the `countCars`, you must write a simple traffic simulator. Imagine you have a four-way intersection controlled by a traffic light. The light is either green in the north/south direction or it is green in the east/west direction.

Initially, the north/south direction is green. The first function parameter is a tuple of two numbers, where the first is the amount of time that the north/south stays green, and the second is the amount of time that the east/west direction stays green.

The second input is a 4-tuple of integer lists. These lists tell you how many cars reach the intersection from each direction (order is north-south-east-west) every 15 seconds, starting at time t=0, then t=15, and so on. You should treat the first value in each list as already waiting at the light at t=0.

The final input is the amount of time (in seconds) that you're simulating. The largest value in the test cases is only around 60 seconds (one minute).

Whenever a light turns green, it takes the *first* car **two seconds** to cross the intersection, and then subsequent cars lined up will only take **one second**.

For example, suppose the north/south light turns green at t=10 and then will turn red at t=20. The first car reaches the end of the intersection at t=12, the second at t=13, and so on. So if there is a full line of cars going north, then 9 of them will go through on that cycle (and the same number will proceed south if there's a full line in *that* direction).

Your job is to calculate how many cars will make it through the intersection in the allotted time.

Simple Pacman

Imagine we have a very simple sort of Pacman game, where our character starts in the bottom left of a 10x10 grid with no walls, and wants to reach the top right corner. There are three ghosts who try to stop him, and a couple special powers that help Pacman. The game advances in discrete time steps. First, the ghosts move, then Pacman moves. Here are the rules for how each character moves:

Ghost Moves
There are 3 ghosts, Blinky, Inky, and Pinky. Blinky will try to chase Pacman by first matching his vertical position (column) and then moving down/up the column to Pacman's row. Inky will do the reverse, matching Pacman's row and then chasing to his column. Pinky will move to Pacman's location if he is adjacent. Otherwise, Pinky will follow a patrol near the goal point, pictured below.

```
.......v<.
.......v^<
.......>>^
..........
..........
..........
..........
..........
..........
..........
```

Pacman Moves

Pacman moves either up or right towards his goal. He chooses whichever direction requires more travel, favoring "right" if there's a tie. He will never move on or next to a ghost though, even if this means choosing the less favored direction. Pacman can also use a **boost**, teleporting to whichever square within two spaces (including diagonally) is closest to the goal.

However, he will only use this boost if his regular moves go on top of a ghost or next to a ghost. Pacman must make 5 moves before using the boost again. Pacman will immediately win upon reaching the goal square, and loses immediately if moving on top of a ghost or if a ghost moves on top of him.

Your job is to fill in `solvePacman`. This function takes three coordinates for the initial positions of the ghosts, Blinky, Inky, and Pinky (Pinky will always start in one of its "patrol" locations). Determine if Pacman will reach the goal or not following these rules.

For example, given this start configuration (M is "Pacman", P is "Pinky"):

```
.......P..
...B......
..........
..........
..........
..........
........I.
..........
..........
M.........
```

The way this scenario evolves, we reach this position after the eighth move:

```
.......P..
..........
..........
..........
....B.....
....MI....
..........
..........
..........
..........
..........
```

Pacman uses the boost here to get around the ghosts, so we end up here (Blinky and Inky are on top of one another:

```
..........
.......P..
..........
......M...
.....B....
..........
..........
..........
..........
..........
..........
```

However, after 3 more moves, Pacman will be cornered:

```
..........
..........
.........P
.......I.M
........B.
..........
..........
..........
..........
..........
```

Since Pacman can't use his boost on this step, he will be caught, so your function should return `False`. Here's an example that will return `True`:

```
..........
..........
........P.
..........
..........
B.........
..........
..........
..........
M...I.....
```

Now Pacman will use the boost early enough so that it's ready again to jump past Pinky to the end of the grid.

A Regular Expression

Your job is to fill in `solveRegex` using your `runFSM` function. This function should evaluate if the input string satisfies the regular expression `(abc)+d[e-g]*`. That is, there should be one or more instances of the triplet `abc`, followed by the letter `d`, and then any number of characters that are `e`,`f`, and `g`. The input strings will only consist of the seven letters `a-g`.

You should use the `RegexState` type as an enumeration of the states your FSM can be in. You should use `RegexFailure` as one of your states, but obviously you must add others.

A Context-Free Grammar

Now fill in `solveCFG`. This should use `runPDA` to determine if the input string belongs to a context-free-grammar we may describe as $a^n b^m c^m d^n$. There should be `n` instances of the character `a` (`n` is any non-negative number), followed by `m` instances of the character `b` (`m` is a potentially different non-negative number), and then `m` instances of `c`, and finally `n` instances of the letter `d`. The input string will only have the characters `a-e`.

You should use the `CFGState` enum type to capture the states of the PDA. Once again, we've provided some starting states: `CFGSuccess`, `CFGFailure`, and `CFGInitial`.

Test your answers by running:

```
$ sh-test 1-10
```

Lecture 11 - Module Review & Practice Problems

We've got some final practice problems for this module in `M1Lecture11.hs`. These problems are a bit more challenging on average, so I've included `MonadLogger` in all of them so you can use log statements to help debug.

Count Intervals Excluding Spikes

For the function `countIntervalsExcludingSpikes`, you're given a list of inclusive integer intervals. You would like to count the number of spaces occupied by all the intervals. By inclusive, we mean that the interval occupies both its start and end point. So the interval `(1,3)` counts for three spaces. The collective group `(1,3)`, `(5,8)` and `(6,10)` includes 9 spaces.

However, the second input is a list of "spikes". Spikes can break up intervals and are thus excluded from the count. Any number of spikes can be in an interval, and some spikes do not fall in any interval! So given the above set of 3 intervals and the spike list `[6, 8, 12]`, your final answer should be 7.

Note: The area covered by the intervals can be very large (millions of spaces), but the number of intervals will be relatively small. Thus you should take care that your solution's runtime depends on the number of intervals rather than the number of spaces they cover.

Backpack Meals

You're going on a day-long hiking trip and you want to pack two meals into your backpack, a breakfast, and a lunch. You have an assortment of breakfast options and a separate assortment of lunch options. You must take one of each. Seeing as how it's a strenuous hike, you want to get the most calories by packing the *heaviest* options you can. However, you are limited in the weight you can carry.

In the function `backpackMeals`, your inputs are a list of the weights of the breakfast options, a list of the lunch options, and the total weight you can carry. You should return an integer with the highest total food weight that will fit into your backpack by choosing one of each option. You can return 0 if you cannot fit any pairs into your pack.

Stack Boxes

Your first input is a list of strings, where each character represents a labeled box. Your second input is a list of commands for moving boxes from one stack to another. These tuples have 3 tuples. The first is the source stack (0-indexed), the second is the destination stack, and the third is the number of boxes to move. You want to execute each of these move commands to create a new series of stacks. You should then return the string made by concatenating the final stacks in order.

The boxes are light, but a bit unwieldy. This means you can carry up to **two at a time**. So if your command tells you to move 3 boxes, you'll move the first two on top of the stack, and then place the third box on top of the new stack. (If too many boxes are requested simply move all the boxes that are on the source stack). Here's an example:

```
stackBoxes ["abc", "", "defgh", "ij"] [(0,1,1), (2,1,3), (3, 2, 2)]
```

Here's the initial look of the stacks:

```
    d
    e
a   f
b   g i
c   h j
0 1 2 3
```

After the first move, it looks like this:

```
  d
  e
  f
b g i
c a h j
0 1 2 3
```

For the second move, we move `d` and `e` together, and finally place `f` on top.

```
  f
  d
b e g i
c a h j
0 1 2 3
```

For the final move, we move `i` and `j` together.

```
  f i
  d j
b e g
c a h
0 1 2 3
```

Your final result then should be the string `"bcfdeaijgh"`.

Run Assembly

The inputs to `runAssembly` are a series of commands for a basic assembly program that uses 3 registers (`eax`, `ebx`, and `ecx`), and these are represented with an enumerated type. Here are the commands you can use:

`LoadValue` - This overwrites a register's value with the given number.
`AddRegister` - This adds the value of the second given register to the first register
`SubRegister` - This subtracts the value of the second given register to the first register
`MultRegister` - This multiples the value of the second given register to the first register
`PrintRegister` - This "prints" the value in the given register to the screen.

Don't worry about the mechanics of actually printing values. Your only job is to return a list of all the integers that are printed by the program, as represented by the list of input commands.

Word Search

In `wordSearch`, your input is a nested list of characters (or alternatively, a list of strings). You are guaranteed that each sublist is the same length. Determine if the input word can be found by taking a **straight line** in any of the 8 directions (up/down/left/right and their combinations). Here's an example input grid with some valid words highlighted:

a **g** **h** **i** **p**
g **p** **g** o **r**
z u **p** **e** **y**
m c s **l** q
h e v l **e**

Example words that should return `True` include **`apple`**, **egg**, **hip**, and **pry**. If your input is, for example, `hop`, it would return `False` with this grid.

Hidden Message

Now for another type of search. In `hiddenMessage`, you're trying to decode a message. Your inputs are a series of strings where each string contains letters from the final message that are in order, but incomplete. Of course, every letter in the final word will appear in at least one of the input strings. You are also guaranteed that the final word does not have any duplicate letters.

Return the hidden message word, or return `Nothing` if the strings do not provide sufficient logic clues for you to figure it out. Here's an example where the hidden word is `"winter"`:

```
hiddenMessage ["wne", "witr, "ine", "int", "ter"]
```

There are various ways to optimize this solution, but most of them use data structures that we'll learn in module 2. The test cases are all small scale, so don't worry too much about optimizing!

Test your answers by running:

```
$ sh-test 1-11
```