Advent of Code 2024

Literate Clojure solutions to programming puzzles.

Rens Oliemans

December 10, 2024

Contents

Finding the Chief Historian! This program contains my Advent of Code solutions for 2024, which you can find on my sourcehut and GitHub. I believe

GitHub doesn't show the results of code blocks, which means that viewing it there might leave you a bit confused.

In general, I've added line numbers to code blocks when that code block is part of the solution file. In some cases, I've added some code that explains, clarifies, justifies or otherifies something. Those lines aren't numbered if they aren't necessary to the final solution.

Utils

I define some common functions in `aoc.util`, mostly related to parsing the input. The input always comes in a file but also usually has an example input. The former is a file (which we read as a string with `slurp`) and the latter is just a string in the same format. Therefore it's easiest to let the days itself take care of reading the file (since they also have the example input), and just operate on strings here.

```
(ns aoc.util
  (:require [clojure.string :as str]))

(defn string-as-lines
  "Outputs the string as a vector, one element per line."
  [input]
  (str/split input #"\n"))
```

Often the lines contain numbers:

```
(defn string-as-numbers-per-line
  "Assumes there is a number on each line: we parse it and return a
  vector, one element per line. Technically each number is parsed with
  `read-string`, so it isn't just limited to numbers, but I've only
  tested numbers."
  [input]
  (let [lines (string-as-lines input)]
    (map read-string lines)))
```

or lists of numbers. This was the case in both Day 1 and Day 2, where the input had the following format:

```
7 6 4 2 1
1 2 7 8 9
9 7 6 2 1
1 3 2 4 5
8 6 4 4 1
1 3 6 7 9
```

Here, we want the whole file to be represented by a vector, where each element is itself a vector of the space-separated numbers on a line.

```
(defn num-list-per-line
  "Returns a vector of vectors, the outer vector has an element per
  line, the inner has space-separated elements. "
  [input]
  (let [lines (string-as-lines input)]
```

```
20      (->> lines
21           (map #(str/split % #"\s+"))
22           (map #(map read-string %)))))
```

This function, run on the example table seen above, will return the following:

```
(num-list-per-line example)
```

```
((7 6 4 2 1) (1 2 7 8 9) (9 7 6 2 1) (1 3 2 4 5) (8 6 4 4 1) (1 3 6 7 9))
```

1 Day 1

1.1 Part 1

We need to reconcile two lists. We get them in the following form:

```
3   4
4   3
2   5
1   3
3   9
3   3
```

And our goal is to find the "distance" between the two lists.

> To find the total distance between the left list and the right list, add up the distances between all of the [sorted] pairs you found.

For the example above, the correct answer is **11**.

My strategy is: convert the input to pairs of numbers (`aoc.util/num-list-per-line` takes care of this), transpose them (so we have two lists), sort them, transpose them again (pairs), and take the difference and sum it. Makes sense? We need the two tiny helper functions `sum` and `transpose`:

```
(ns aoc.1)

(defn sum "Finds the sum of a vector of numbers" [vec]
  (reduce + vec))

(defn transpose "Transposes a matrix" [m]
  (apply mapv vector m))
(defn p1 [input]
  (let [input (aoc.util/num-list-per-line input)]
    (->> input
         (transpose)
         (map sort)
         (transpose)
         (map #(abs (- (first %) (second %))))
         (sum))))
```

It works for the testinput, fantastic. Now let's open the file and run it on the input. The input file for day 1 can be found in the file `inputs/1`.

```
(assert (= 11 (p1 testinput)))
(def input (slurp "inputs/1"))
(p1 input)
```

```
2057374
```

Hurrah! We get a **Gold Star**!

1.2 Part 2

Now, we need to find a "similarity score" for the two lists:

> Calculate a total similarity score by adding up each number in the left list after multiplying it by the number of times that number appears in the right list.

A naive way to do this would be to iterate over the first list, where, for each element, we count how many items in the second list are equal to that element, and multiply the element with the count. However, you'd be doing a lot of duplicate counting. A faster way to do it is to convert the second (it doesn't really matter which one you pick) list to a map once, with `{element frequency}`. Let's use the function `frequencies`!

```
(frequencies (last (transpose (aoc.util/num-list-per-line testinput))))
```

```
{4 1, 3 3, 5 1, 9 1}
```

Now, we can iterate over the first list (which we get by `(transpose (numbers input))`), multiply the element itself by the count in `frequencies`, and sum the result.

```
18 (defn p2 [input]
19   (let [input (transpose (aoc.util/num-list-per-line input))
20         one (first input)
21         freqs (frequencies (second input))]
22     (->> one
23          (map #(* % (freqs % 0)))
24          (sum))))
25
26 (assert (= 31 (p2 testinput)))
27 (p2 input)
```

```
23177084
```

2 Day 2

2.1 Part 1

Analysing some unusual data from a nuclear reactor. The data consists of *reports* separated by lines, each of which is a list of numbers (*levels*), separated by spaces.

```
7 6 4 2 1
1 2 7 8 9
9 7 6 2 1
1 3 2 4 5
8 6 4 4 1
1 3 6 7 9
```

We need to find out how many reports are **safe**, which is the case if all levels are gradually increasing or decreasing. This is defined as such:

> [A] report only counts as safe if both of the following are true:
>
> - The levels are either all increasing or all decreasing.
> - Any two adjacent levels differ by at least one and at most three.

In the example input, there are **2** safe reports—the first and last.

Let's convert all numbers to the difference between the previous number. Then, a report is safe is all numbers are of the same sign, and the absolute of the number is between 1 and 3.

Since we're computing the difference between each element and the element before, I want to use `partition`, which does exactly this. Then, we can use `mapv` to compute the difference. For the last element of the testinput:

```
1 (ns aoc.2)

2 (defn diffs [record]
3   (->> record
4        (partition 2 1)
5        (mapv (fn [[a b]] (- b a)))))
```

test it out:

```
(diffs (last (aoc.util/num-list-per-line testinput)))
```

```
[2 3 1 2]
```

Now just use that to determine whether a record is safe. Recall that the testinput had **2** safe records.

```
(defn is-safe? [record]
  (let [differences (diffs record)]
    (and (every? #(<= 1 (abs %) 3) differences)
         (apply = (map pos? differences)))))

(defn p1 [input]
  (->> (aoc.util/num-list-per-line input)
       (filter is-safe?)
       (count)))

(p1 testinput)
```

```
2
```

```
(def input (slurp "inputs/2"))
(p1 input)
```

```
242
```

2.2 Part 2

> Now, the same rules apply as before, except if removing a single level from an unsafe report would make it safe, the report instead counts as safe.

First I had a smart idea. Check out e2dcab2f0de76c21477c5e871e029f0282c8fabc. It is much more efficient than the current solution, but much more convoluted and ugly to read. Right now, I just remove each level one by one and check if the record is safe then.

```
(defn drop-nth [coll n]
  (keep-indexed #(if (not= %1 n) %2) coll))

(defn dampened-is-safe? [record]
  (some is-safe? (map #(drop-nth record %)
                      (range (count record)))))

(defn p2 [input]
  (->> (aoc.util/num-list-per-line input)
       (filter dampened-is-safe?)
```

```
29          (count)))
30
31  (p2 testinput)
```

```
4
```

```
32  (p2 input)
```

```
311
```

2.2.1 Benchmark results

The old solution took on average `3.8` milliseconds to execute `(p2 input)`, and the new solution about `6.0`. This is worth it, IMO, since the code is *much* simpler. Next time, first do the easy thing, and then benchmark to see if it needs to be improved!

3 Day 3

3.1 Part 1

We have an input string that contains a lot of characters, for example:

```
xmul(2,4)%&mul[3,7]!@^do_not_mul(5,5)+mul(32,64]then(mul(11,8)mul(8,5))
```

The goal is to extract all substrings that are of the exact form `mul(\d+,\d+)`, and in that case multiply the two numbers together. This is straightforward, I'm not really going to create any helper functions: parse with regex, convert to int, multiply and sum.

```
(ns aoc.3)

(defn p1 [input]
  (let [matches (re-seq #"mul\((\d+),(\d+)\)" input)]
    (->> matches
         (map #(list (Integer/parseInt (nth % 1)) (Integer/parseInt (nth % 2))))
         (map #(apply * %))
         (reduce +))))

(let [input (slurp "inputs/3")]
  (p1 input))
```

```
155955228
```

3.2 Part 2

We get a new example string for Part Two:

```
xmul(2,4)&mul[3,7]!^don't()_mul(5,5)+mul(32,64](mul(11,8)undo()?mul(8,5))
```

This contains the substrings `don't()` and `do()`, which disable and enable `mul()` instructions. I can do fancy clojure things, but Emacs is way too good for this, so let's do it quickly in Elisp. We want to remove everything from the input file that's in between a `don't()` and a `do()` instruction, and then call `(p1)` on this input. There are three slightly tricky things about this:

- The input file has some newlines, and in some cases a `do()` instruction is on a later line than the previous `don't()` instruction.
- You need to match non-greedy in between a `don't()` and a `do()`.
- If you call `(replace-regexp)` with just the regex and replacement string, it will move point to the last match. This is easily fixed by adding the fourth and fifth arguments to `replace-regexp`: `START` and `END`.

So, here's some elisp code that does that.

```
(with-temp-buffer
  (insert-file-contents "inputs/3")
  (replace-regexp "\n" "" nil (point-min) (point-max))
  (replace-regexp "don't().+?do()" "" nil (point-min) (point-max))
  (write-region (point-min) (point-max) "inputs/3-enabled"))
```

And back to clojure for the now trivial second part.

```
(let [fixed-input (slurp "inputs/3-enabled")]
  (p1 fixed-input))
```

```
100189366
```

4 Day 4

4.1 Part 1

We need to find all instances of `XMAS`, appearing in a text like below, either horizontally, vertically, or diagonally, including written backwards. According to these rules, the example below contains **18** `XMAS`-es.

```
MMMSXXMASM
MSAMXMSMSA
AMXSXMAAMM
MSAMASMSMX
XMASAMXAMM
XXAMMXXAMA
SMSMSASXSS
SAXAMASAAA
MAMMMXMMMM
MXMXAXMASX
```

My idea is to search on the letter `X` and use each `X` as a starting point, where we count `XMAS` occurrences in each of the 8 different directions. Let's create a function `count-xmases-at` that counts the number of `XMAS`-es starting from a location. Then simply call that for each `X` found in the grid and sum.

We're going to need to define two helper functions:

`char-locations` Returning all the locations of given character in the grid;

`is-xmas?` A function that takes the `grid`, a `start` coordinate and a `direction`. It returns `true` if `"XMAS"` occurs in the `grid` from `start` in the given `direction`.

Since we're working in a grid, let's make `char-locations` return a 2d vector like `[0 0]` to denote the coordinates in the grid. While we're at it, let's define those directions like so:

```
1  (ns aoc.4)

2  (def directions
3    [[-1  0]   ; Up
4     [ 1  0]   ; Down
5     [ 0 -1]   ; Left
6     [ 0  1]   ; Right
7     [-1 -1]   ; NW
8     [-1  1]   ; NR
9     [ 1 -1]   ; SW
10    [ 1  1]]) ; SE
```

Which is a nice format to have, because you can then do something like this:

```
(defn move [start direction]
  (let [[x y] start
        [x' y'] direction]
    [(+ x x') (+ y y')]))

(let [start [4 5]
      direction [-1 0]] ;; up
  (move start direction))

[3 5]
```

Which is really nice! So, let's define the final functions necessary for Part One:

```
(defn char-locations [grid x]
  (mapcat
   (fn [row string]
     (keep-indexed (fn [col char] (when (= char x) [row col]))
                   string))
   (range)
   grid))

(defn is-xmas? "Does the grid `grid` contain the string \"XMAS\",
 starting at `start` and going in `direction`?"
  [grid start direction]
  (let [[startx starty] start
        [x' y'] direction]
    (loop [x startx
           y starty
           chars (seq "XMAS")]
      (if (empty? chars)
        true
        (if (not= (get-in grid [x y]) (first chars))
          false
          (recur (+ x x')
                 (+ y y')
                 (rest chars)))))))
```

Now we can tie everything together. `is-xmas?` returns true if the grid contains the word `"XMAS"` in a given `direction`. After we've found all `X` characters, we can count the amount of `XMAS`-es connected to it by counting all `direction` for which `is-xmas?` returns true.

```clojure
(defn count-xmases-at [grid start directions]
  (count (filter #(is-xmas? grid start %) directions)))

(defn p1 [input]
  (let [grid (aoc.util/string-as-lines input)
        xs (char-locations grid \X)]
    (->> xs
         (map #(count-xmases-at grid % directions))
         (reduce +))))

(assert (= 18 (p1 example)))
(def input (slurp "inputs/4"))
(p1 input)
```

```
2447
```

4.2 Part 2

Ah, it seems the Elf thinks we're idiots because they use letters more literally. We don't need to find the string `XMAS`, we need to find the string `MAS` in an X, like so!

```
M.S
.A.
M.S
```

We could have reused the functionality above to search for `MAS`-es, and then only count a `MAS` that has a nice diagonal partner sharing the `A`. However, I found that a bit tricky to reason about, so I've opted to search for all of the `A`-s in the text, and finding `MAS` strings diagonally from that `A`. If there are exactly two `MAS`-es, we know that we got an `X-MAS`.

Instead of `is-xmas?`, we now have `is-mas?`, checking from a `middle A` instead of a starting `X`. Note that we're only counting **X**-`MAS`-es, so only use diagonals:

```clojure
(def diagonal-directions
  [[-1 -1]   ; NW
   [-1  1]   ; NE
   [ 1 -1]   ; SW
   [ 1  1]]) ; SE
```

`is-mas?` is now pretty trivial:

```clojure
(defn is-mas? [grid middle direction]
  (let [[x y] middle
```

```
54          [x' y'] direction]
55      (and (= \M (get-in grid [(+ x x') (+ y y')]))
56           (= \S (get-in grid [(- x x') (- y y')])))))
```

And `count-mases-at` is virtually identical to `count-xmases-at` from Part One.

```
57  (defn count-mases-at [grid middle directions]
58    (count (filter #(is-mas? grid middle %) directions)))
59
60  (defn p2 [input]
61    (let [grid (aoc.util/string-as-lines input)
62          as (char-locations grid \A)]
63      (->> as
64           (map #(count-mases-at grid % diagonal-directions))
65           (filter #(= % 2))
66           (count))))
67
68  (assert (= 9 (p2 example)))
69  (p2 input)
```

```
1868
```

5 Day 5

5.1 Part 1

Graphs! We get an input file that looks like this:

```
47|53
97|13
97|61
97|47
75|29
61|13
75|53
29|13
97|29
53|29
61|53
97|53
61|29
47|13
75|47
97|75
47|61
75|61
47|29
75|13
53|13

75,47,61,53,29
97,61,53,29,13
75,29,13
75,97,47,61,53
61,13,29
97,13,75,29,47
```

The first part contains required orderings, where `29|13` means that `29` should always come before `13`. The second part contains "updates" that might or might not be correctly sorted. In Part One, we need to take the correctly sorted updates, take the middle number, and sum those. I wonder what the second part will be? Actually, I don't wonder, I'm virtually certain of it so I'm just going to sort them already. If the update is equal to the sorted input, it's sorted and we can solve Part One.

I already alluded to graphs, that's because you can think of this as a ~~DAG~~ Directed Graph. In the case before, `29|13` will lead to a vertex from `29` to `13`. My "graph" will basically be a list of dependencies, but I'll call it a

graph because that's cool and it sort of is one. Before we get into the weeds, let's zoom out and think of what we need: the sum of the middle numbers of the sorted updates.

First look at the easy functions, leaving `sort` and `build-dependency-graph` empty for the time being:

```
(ns aoc.5
  (:require [clojure.string :as str]))

(defn sort [dependency-graph update])
(defn build-dependency-graph [orderings])

(defn sorted? [dependency-graph update]
  (= update (sort dependency-graph update)))

(defn middle-num
  "Finds the middle string in a list of string, and parses it to a
  number. Assumes the length of the list list is odd."
  [update]
  (read-string (nth update (/ (count update) 2))))
```

Now we can write `p1`. Since I expect to need the orderings, updates and dependency-graph later as well, I'll create a small function `parse-input` that extracts these from the puzzle input.

```
(defn parse-input
  "Parses an input string and returns three useful objects.
  The first obj is a list of orderings, strings of type \"A|B\".
  The second obj is a list of updates, each one a list of strings.
  The third obj is a dependency graph, a map."
  [input]
  (let [[orderings updates] (str/split input #"\n\n")
        orderings (str/split orderings #"\n")
        updates (str/split updates #"\n")
        updates (map #(str/split % #",") updates)
        dependency-graph (build-dependency-graph orderings)]
    [orderings updates dependency-graph]))

(defn p1 [input]
  (let [[orderings updates dep-graph] (parse-input input)
        sorted? (partial sorted? dep-graph)]
    (->> updates
         (filter sorted?)
         (map middle-num)
         (reduce +))))
```

Hmm, yes, extremely reasonable, but we haven't yet filled in `build-dependency-graph` and `sort`. `build-dependency-graph` should take as input the `orderings` (a list of strings from the input, separated by `|`), and return a map of the following form:

```
{"75" ["97"], "13" ["97" "61" "29" "47" "75" "53"], ...}.
```

To do so, I'll first create a hash-map of the following form:

```
{"75" ["97"], "13" ["97"], "13" ["61"], ...},
```

and then merge identical keys with `merge-with` and `into`, creating our desired dependency graph.

```
35 (defn build-dependency-graph
36   [orderings]
37   (let [order-pairs (->> orderings
38                          (map #(str/split % #"\|"))
39                          (map #(hash-map (second %), [(first %)])))]
40     (apply (partial merge-with into) order-pairs)))
```

Verifying that this next result is correct is left as an exercise for the reader, but let's test it out on the example input:

```
(let [[orderings _ _] (parse-input example)]
  (build-dependency-graph orderings))

{"61" ["97" "47" "75"],
 "47" ["97" "75"],
 "53" ["47" "75" "61" "97"],
 "13" ["97" "61" "29" "47" "75" "53"],
 "75" ["97"],
 "29" ["75" "97" "53" "61" "47"]}
```

And now, ladies and gentleman, the moment you've all been waiting for, `sort`! We need to sort an `update` based on a `dependency-graph`. You can see it below, but how it works:

1. It creates a `graph`: a subset of `dep-graph`, *limited to the items local to the current* `update`. It starts with an empty map `{}`, and then for each `item` in `update`, adds the elements in the `dependency-graph` that depend on `item`. `graph` ends up as a map with key a number, and value a set of the dependencies.

   Limiting the dependency graph to be local only to the current `update` gives us a tremendous advantage: we can sort the items based on the number of dependencies each item has.

2. Sort the items in `update` by their amount of dependencies.

```clojure
(defn sort
  "Sort a list of strings based on a dependency map.
   The map defines which elements should come after others."
  [dep-graph update]
  (let [graph (reduce (fn [acc item]
                        (assoc acc item
                               (set (get dep-graph item []))))
                      {} update)
        local-deps (fn [deps] (filter #(contains? (set update) %) deps))]
    (vec (sort-by (fn [item]
                    (let [deps (get dep-graph item [])]
                      (count (local-deps deps))))
                  update))))
```

Now we got everything, ain't we?

```clojure
(assert (= 143 (p1 example)))
(def input (slurp "inputs/5"))
(p1 input)
```

```
4637
```

yes

5.2 Part 2

Surprise surprise, we need to sort the incorrect updates! We need to take the sum of the middle numbers of only the *incorrect* updates. Our prescience is immeasurable.

```clojure
(defn p2 [input]
  (let [[orderings updates deps] (parse-input input)
        is-sorted? (partial sorted? deps)
        sort (partial sort deps)]
    (->> updates
         (filter #(not (is-sorted? %)))
         (map sort)
         (map middle-num)
         (reduce +))))

(assert (= 123 (p2 example)))
(p2 input)
```

```
6370
```

6 Day 6

6.1 Part 1

We get a grid again, now representing a map. It looks like this:

```
....#.....
.........#
..........
..#.......
.......#..
..........
.#..^.....
........#.
#.........
......#...
```

The `^` represents the starting location of our guard, and they start by going *up*. A `#` is an obstacle, and will force the guard to move direction, turning 90° clockwise. Our goal is to find out how many distinct places the guard has entered by the time he leaves the puzzle.

If you replace entered places by `X`, you'd get the following output, with **41** distinct places:

```
....#.....
....XXXXX#
....X...X.
..#.X...X.
..XXXXX#X.
..X.X.X.X.
.#XXXXXXX.
.XXXXXXX#.
#XXXXXXX..
......#X..
```

Turning clockwise means that we have only four directions:

```
1 (ns aoc.6)

2 (def directions
3   [[-1  0]   ; Up
4    [ 0  1]   ; Right
5    [ 1  0]   ; Down
6    [ 0 -1]]) ; Left
```

Our function will simply compute the route the guard takes as a vector of coordinates, and count the distinct elements of said vector:

```
(defn guard-route
  "Takes a `grid` as input returns a vector of 2d coordinates: the route
  of the guard, starting at `start` and turning clockwise at \"#\"
  characters. "
  [grid start])

(defn p1 [input]
  (let [grid (aoc.util/string-as-lines input)
        start (first (aoc.4/char-locations grid \^))
        route (guard-route grid start)]
    (count (distinct route))))
```

As for `guard-route`, we loop through the grid, where each iteration of the loop is a move: go to the next location given some direction, or change direction, building a `route` along the way. We replace the `^` character with a `.` after determining the start so that we only have two cases to deal with, `.` and `#`. We can reuse the `char-locations` formula from Day 4 (which gives us a list of coordinates where a certain character can be found) to find our starting location.

```
(defn replace-char
  [grid [x y] new-char]
  (update grid x
          #(str (subs % 0 y)
                new-char
                (subs % (inc y)))))

(defn guard-route [grid start]
  (let [size (count grid)
        grid (replace-char grid start \.)]
    (loop [location start
           directions (cycle directions)
           route []]
      (let [[x y] location
            [x' y'] (first directions)
            next-location [(+ x x') (+ y y')]
            next-object (get-in grid next-location)
            route (conj route location)]
        (condp = next-object
          nil route
          \. (recur next-location
                    directions
                    route)
          \# (recur location
```

```
                    (next directions)
                    route))))))
```

Perhaps this is a little too imperative, but I'm fine with it.

```
(assert (= 41 (p1 example)))
(def input (slurp "inputs/6"))
(p1 input)
```

```
5208
```

6.2 Part 2

It's of course possible that the guard enters a loop, but fortunately that didn't occur in the input we were given. Part Two is concerned with *creating* loops by adding obstacles. Specifically, *how many loops can we create by adding just a single obstacle?*

I'm afraid that I'll have to create a very similar function to `guard-route`, except that now we keep track of the places we've been before. If we ever enter the same location while going in the same direction, we know we've entered a loop and can exit immediately. In that case, let's return `true` and name the function `route-has-loop?`. Since we're exiting earlier and I don't want to create cycle-detection, I'm not reusing the function from Part One. In python I'd use a generator, but I haven't figured out `lazy-seq` yet in clojure.

I can't think of a way to do this intelligently, but at least one insight is that you don't have to consider *all* cases: you only have to add obstacles on parts of the original route; adding them elsewhere will have no effect.

```
(defn route-has-loop? [grid start])

(defn p2 [input]
  (let [grid (aoc.util/string-as-lines input)
        start (first (aoc.4/char-locations grid \^))
        route (disj (set (guard-route grid start)) start)]
    (->> route
         (pmap (fn [new-obstacle]
                 (route-has-loop? (replace-char grid new-obstacle \#) start)))
         (filter true?)
         (count))))
```

`route-has-loop?` is virtually identical to `guard-route`, except that we keep track of the `visited` set (keeping track of visited `[location direction]` pairs), and that we return `true` or `false` instead of the route.

```
58 (defn route-has-loop? [grid start]
59   (let [size (count grid)
60         grid (replace-char grid start \.)]
61     (loop [location start
62            directions (cycle directions)
63            visited #{}]
64       (let [[x y] location
65             [x' y'] (first directions)
66             next-location [(+ x x') (+ y y')]
67             next-object (get-in grid next-location)
68             pair [next-location [x' y']]]
69         (if (contains? visited pair)
70           true ;; we have a loop!
71           (condp = next-object
72             nil false ;; we exited the puzzle
73             \. (recur next-location
74                       directions
75                       (conj visited pair))
76             \# (recur location
77                       (next directions)
78                       (conj visited pair)))))))))
```

On my laptop, this takes about 15 seconds to run on a single thread, but by default uses all of the threads (just by changing `map` into `pmap`, how freaking awesome is that!)

```
79 (assert (= 6 (p2 example)))
80 (p2 input)
```

```
1972
```

7 Day 7

7.1 Part 1

The elephants stole our operators! We had a list of equations, but they stole the operators between the numbers. We get an input where each line represents a single equation, which may be correct. We have to determine whether the equation can be correct, if we limit ourselves to `+` and `*`. In this example:

```
190: 10 19
3267: 81 40 27
83: 17 5
156: 15 6
7290: 6 8 6 15
161011: 16 10 13
192: 17 8 14
21037: 9 7 18 13
292: 11 6 16 20
```

only three of the equations can be made true, and their results sum up to **3749**—that is our goal.

```
(ns aoc.7 (:require [clojure.string :as str]))
```

Again we get a familiar pattern: `map`, `filter`, `reduce`.

```
(defn is-correct? [equation])

(defn parse-equations [input]
  (let [lines (str/split input #"\n")
        equations (map #(str/split % #": ") lines)]
    (map (fn [[lhs rhs]]
           [(read-string lhs) (vec (map read-string (str/split rhs #" ")))])
         equations)))

(defn p1 [input]
  (->> input
       (parse-equations)
       (filter is-correct?)
       (map first)
       (reduce +)))

(parse-equations example)
```

```
([190 [10 19]]
 [3267 [81 40 27]]
 [83 [17 5]]
 [156 [15 6]]
 [7290 [6 8 6 15]]
 [161011 [16 10 13]]
 [192 [17 8 14]]
 [21037 [9 7 18 13]]
 [292 [11 6 16 20]])
```

Now the banger `is-correct?`. There are ~800 equations, the longest one has 12 numbers to add or multiply, so 4096 possible operations to check out. I think brute-forcing is pretty viable.

```
(defn possible-ops
  [x y]
  [(* x y)
   (+ x y)])

(defn equation-possibilities
  [target nums]
  (->> (range 1 (count nums))
       (reduce (fn [possible-results idx]
                 (->> possible-results
                      (mapcat (fn [result]
                                (possible-ops result (nth nums idx))))))
               [(first nums)])))

(defn is-correct? [equation]
  (let [[result numbers] equation
        targets (equation-possibilities result numbers)]
    (some #(= % result) targets)))

(assert (= 3749 (p1 example)))
(def input (slurp "inputs/7"))
(p1 input)
```

```
12839601725877
```

7.2 Part 2

```
(defn possible-ops
  [x y]
  [(* x y)
```

```
41      (+ x y)
42      (Long/parseLong (str x y))])
43
44  (assert (= 11387 (p1 example)))
45  (p1 input)
```

```
149956401519484
```