Advent of Wisp Code 2021

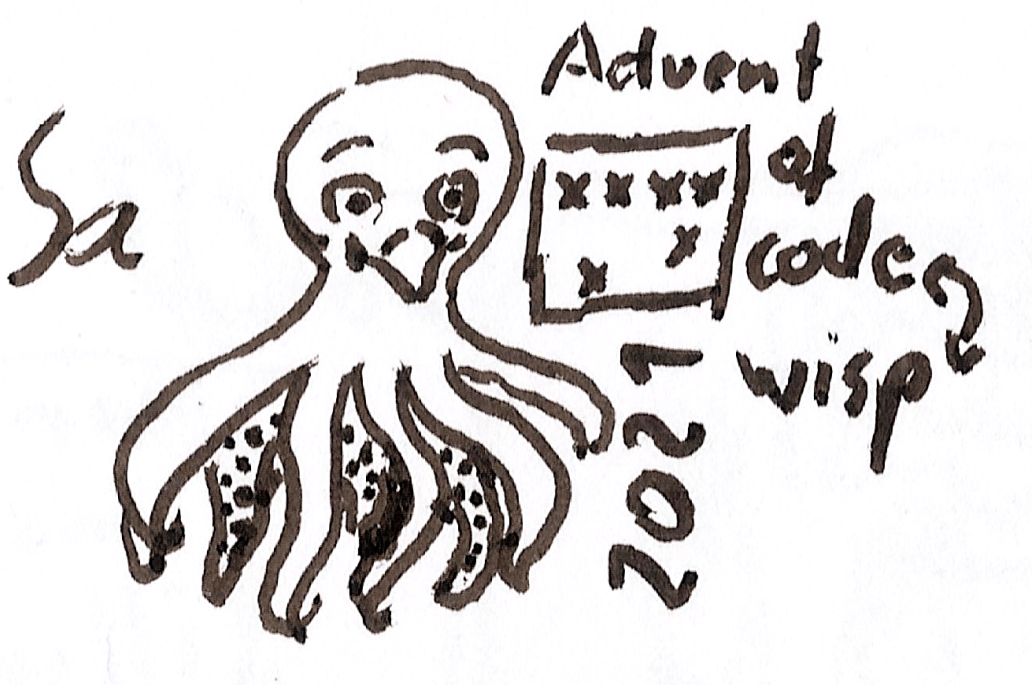
Dr. Arne Babenhauserheide

<2021-12-01 Mi>

Taking part in the advent of code to relax as much as I find time to do. I'll use Wisp.

Check the RSS-Feed to get informed when I solve puzzles.

I will most likely do them with one day delay, because I want to post the solutions here, and because I work during the day and have family.



Contents

Day 1, puzzle 1: Sweep the deep

Count how often the depths of the ocean increases.

```
import : srfi :1 lists
         srfi :9 records

define example-input ' : 199 200 208 210 200 207 240 269 260 263

;; aggregate using a two number window.
define : count-larger current next count
  + count : if {next > current} 1 0

display
  fold count-larger 0
    ;; dropping the first element of the second list
    ;; this shifts the second element in count-larger by 1 => next
    . example-input
    drop example-input 1
```

For the real calculation, I plugged in the input via `define input '(...)`. Hacky but quick.

Day 1, puzzle 2: Sweep the deep averages

Count how often the three element moving sum of the depth increases.

```
import : srfi :1 lists
         srfi :9 records

define example-input ' : 199 200 208 210 200 207 240 269 260 263

;; aggregate using a 4 number window.
define : count-larger n0 n1 n2 n3 count
  + count : if {(+ n1 n2 n3) > (+ n0 n1 n2)} 1 0

display
  fold count-larger 0
    . example-input
    drop example-input 1
    drop example-input 2
    drop example-input 3
```

I'm not fully happy with this code — it is longer and lomre complex than I'd like it to be. But it solves the problem. For a quick fix it is OK, and the adaption from puzzle 1 to puzzle 2 was easy, which is a good sign.

Day 2, Puzzle 1: Pilot the submarine

Read instructions to find the position when following them.

These look like wisp: I'm trying to turn them into code.

The input is now written to a file:

```
forward 5
down 5
forward 8
up 3
down 8
forward 2
```

```
define horizontal 0
define vertical 0
define-syntax-rule : inc var steps
  set! var {var + steps}
define-syntax-rule : dec var steps
  set! var {var - steps}
define (forward steps) : inc horizontal steps
define (down steps) : inc vertical steps
define (up steps) : dec vertical steps

;; load the input as code
;; load "advent-of-wisp-code-2021-d2p1-real-input.w"
load "advent-of-wisp-code-2021-d2p1-example-input.w"

display {horizontal * vertical}
```

Day 2, Puzzle 2: Aim the submarine

The input is the same, but the code is different.

```
define aim 0
define horizontal 0
define vertical 0
define-syntax-rule : inc var steps
  set! var {var + steps}
define-syntax-rule : dec var steps
  set! var {var - steps}
;; the commands and the presence of aim are all that changes:
define (forward steps)
   inc horizontal steps
```

```
   inc vertical {aim * steps}
define (down steps) : inc aim steps
define (up steps) : dec aim steps

;; load the input as code
;; load "advent-of-wisp-code-2021-d2p1-real-input.w"
load "advent-of-wisp-code-2021-d2p1-example-input.w"

display {horizontal * vertical}
```

I actually like this code quite a bit, and adjusting it from puzzle 1 to puzzle 2 was a breeze. It's still a hack, though ...

Update: simple shell-script

While the previous version is kind of a hack (but one that uses a method I actually use to write games), it would be an even funnier hack to replace the auto-pilot with a simple shell script.

```
export AIM=0
export HORIZONTAL=0
export VERTICAL=0
function inc() {
  export ${1}=$((${1} + ${2}))
}
function dec() {
  export ${1}=$((${1} - ${2}))
}
function forward () {
  inc HORIZONTAL ${1}
  inc VERTICAL $((${AIM} * ${1}))
}
function down () {
  inc AIM ${1}
}
function up () {
  dec AIM ${1}
}
source "advent-of-wisp-code-2021-d2p1-example-input.w"

echo $(($HORIZONTAL * $VERTICAL))
```

Would you bet your life on it? :-)

Day 3, Puzzle 1: Diagnose a Dive

Calculate the most common bit in each position. The resulting bits give the diagnostic number γ. Using least common bit gives ε.

Example Input:

```
00100
11110
10110
10111
10101
01111
00111
11100
10000
11001
00010
01010
```

```
import : only (ice-9 rdelim) read-line

define : split-line-into-numbers line
  map string->number : map string : string->list line

define : map-over-lines fun filename
  let : : port : open-input-file filename
    let loop : (lines '()) (line (read-line port))
      if : eof-object? line
        begin
          close port
          reverse! lines
        loop
          cons : fun line
               . lines
          read-line port

define input
  map-over-lines split-line-into-numbers
    . "advent-of-wisp-code-2021-d3p1-example-input.dat"

define len/2 {(length input) / 2}
define most-common : map (λ(x) (if {x > len/2} #\1 #\0)) : apply map + input

define γ
```

```
  string->number : apply string most-common
                 . 2
define ε
  string->number : apply string : map (λ(x) (if (equal? x #\1) #\0 #\1)) most-common
                 . 2
display {γ * ε}
```

This is more complex than I'd like it to be. The most important missing piece is "read all lines".

Day 3, Puzzle 2: Diagnose for Life

Filter the numbers bit by bit, keeping only those where the bit in the given position is the most common bit. If only one number remains, that's the result.

```
00100
11110
10110
10111
10101
01111
00111
11100
10000
11001
00010
01010
```

```
import : only (ice-9 rdelim) read-line
         srfi :9 records
         only (srfi :26) cut

define : split-line-into-numbers line
  map string->number : map string : string->list line

define : map-over-lines fun filename
  let : : port : open-input-file filename
    let loop : (lines '()) (line (read-line port))
      if : eof-object? line
        begin
          close port
          reverse! lines
        loop
```

```
          cons : fun line
               . lines
          read-line port

define input
  map-over-lines split-line-into-numbers
    . "advent-of-wisp-code-2021-d3p1-example-input.dat"

define : most-common input len/2
  map (λ(x) (if {x >= len/2} 1 0)) : apply map + input
define : least-common input len/2
  map (λ(x) (if {x >= len/2} 0 1)) : apply map + input

define : filt input aggregator bitindex
  define len/2 {(length input) / 2}
  define aggregated : aggregator input len/2
  define : matches pattern bitindex
    equal? : list-ref pattern bitindex
             list-ref aggregated bitindex
  filter : cut matches <> bitindex
         . input

define : select aggregator
  let loop : (input (filt input aggregator 0)) (next-bitindex 1)
    if : = 1 : length input
       car input
       loop : filt input aggregator next-bitindex
            + next-bitindex 1

define : list->decimal list-of-numbers
  string->number
    string-join
      map number->string list-of-numbers
      . ""
    . 2

define oxygen : select most-common
define co2scrub : select least-common

display
  * : list->decimal oxygen
      list->decimal co2scrub
```

Day 4, Puzzle 1: Cheat the Squid

A squid attached to the ship. We need to cheat it in Bingo.

Known numbers that will be drawn, and bingo boards:

```
7,4,9,5,11,17,23,2,0,14,21,24,10,16,13,6,15,25,12,22,18,20,8,19,3,26,1

22 13 17 11  0
 8  2 23  4 24
21  9 14 16  7
 6 10  3 18  5
 1 12 20 15 19

 3 15  0  2 22
 9 18 13 17  5
19  8  7 25 23
20 11 10 24  4
14 21 16 12  6

14 21 17 24  4
10 16 15  9 19
18  8 23 26 20
22 11 13  6  5
 2  0 12  3  7
```

Need to find the sum of all the unmarked fields in the winning board (the first to have one fully marked row or column).

Multiply it with the winning number.

```
import : only (ice-9 rdelim) read-line
         srfi :9 records
         only (srfi :26) cut
         only (srfi :1) every fold list-index

define-record-type <bingo>
  make-bingo numbers boards
  . bingo?
  numbers bingo-numbers bingo-numbers-set!
  boards bingo-boards bingo-boards-set!

define : split-bingo-line line
  if : eof-object? line
    list
    map string->number : delete "" : string-split line #\space
```

```
define bingo
  let : : port : open-input-file "advent-of-wisp-code-2021-d4p1-example-input.dat"
    define numbers :  map string->number : string-split (read-line port) #\,
    ;; skip separator line
    read-line port
    define boards
      let read-board : (boards '())
        if : eof-object? : peek-char port
          reverse boards
          read-board
            cons
              let loop : (board '()) (line (split-bingo-line (read-line port)))
                if : null? line
                  reverse board
                  loop : cons line board
                         split-bingo-line : read-line port
              . boards
    close port
    make-bingo numbers boards

define : play-number number board
  map : λ(x) (map (λ(y) (if (equal? number y) #f y)) x)
      . board

define : board-won? board
  define : row-won? row
    every not row
  if ;; force explicit #t or #f
    or
      member #t : map row-won? board
      member #t : apply map (λ(. x) (row-won? x)) board
    . #t #f

display
  let loop : (boards (bingo-boards bingo)) (numbers (bingo-numbers bingo))
    define played : map (cut play-number (car numbers) <>) boards
    define result : map board-won? played
    cond
      : null? numbers
        . #f
      : member #t result
        let : : winner : list-ref played : list-index (λ(x) x) result
          * : car numbers
```

```
              apply + : apply map (λ(. x) (apply + (delete #f x)))  winner
      else
        loop played
             cdr numbers
```

Day 4, Puzzle 2: Let the squid win

A squid attached to the ship. We need to let it win in Bingo. For sure. So we take the board that wins last.

Known numbers that will be drawn, and bingo boards:

```
7,4,9,5,11,17,23,2,0,14,21,24,10,16,13,6,15,25,12,22,18,20,8,19,3,26,1

22 13 17 11  0
 8  2 23  4 24
21  9 14 16  7
 6 10  3 18  5
 1 12 20 15 19

 3 15  0  2 22
 9 18 13 17  5
19  8  7 25 23
20 11 10 24  4
14 21 16 12  6

14 21 17 24  4
10 16 15  9 19
18  8 23 26 20
22 11 13  6  5
 2  0 12  3  7
```

Need to find the sum of all the unmarked fields in the winning board (the first to have one fully marked row or column).

Multiply it with the winning number.

```
import : only (ice-9 rdelim) read-line
         srfi :9 records
         only (srfi :26) cut
         only (srfi :1) every fold list-index remove

define-record-type <bingo>
  make-bingo numbers boards
```

```
  . bingo?
  numbers bingo-numbers bingo-numbers-set!
  boards bingo-boards bingo-boards-set!

define : split-bingo-line line
  if : eof-object? line
    list
    map string->number : delete "" : string-split line #\space

define bingo
  let : : port : open-input-file "advent-of-wisp-code-2021-d4p1-example-input.dat"
    define numbers :  map string->number : string-split (read-line port) #\,
    ;; skip separator line
    read-line port
    define boards
      let read-board : (boards '())
        if : eof-object? : peek-char port
          reverse boards
          read-board
            cons
              let loop : (board '()) (line (split-bingo-line (read-line port)))
                if : null? line
                  reverse board
                  loop : cons line board
                         split-bingo-line : read-line port
              . boards
    close port
    make-bingo numbers boards

define : play-number number board
  map : λ(x) (map (λ(y) (if (equal? number y) #f y)) x)
      . board

define : board-won? board
  define : row-won? row
    every not row
  if ;; force explicit #t or #f
    or
      member #t : map row-won? board
      member #t : apply map (λ(. x) (row-won? x)) board
    . #t #f

display
  let loop : (boards (bingo-boards bingo)) (numbers (bingo-numbers bingo))
```

```
    define played : map (cut play-number (car numbers) <>) boards
    define result : map board-won? played
    cond
      : null? numbers
        . #f
      : every (cut equal? <> #t) result
        let : : winner : list-ref played 0 ;; just choose the first of the last winne
          * : car numbers
              apply + : apply map (λ(. x) (apply + (delete #f x)))  winner
      else
        loop : remove board-won? played
             cdr numbers
```

The adjustment worked very well: the only changes are in the final let loop:

- replace `loop played` by `loop : remove board-won? played` and
- replace `member #t result` by `every (cut equal? <> #t) result` and
- always take the first of the last winners.

Day 5, Puzzle 1: Sidestep the vents

Draw lines and find meeting points.

```
0,9 -> 5,9
8,0 -> 0,8
9,4 -> 3,4
2,2 -> 2,1
7,0 -> 7,4
6,4 -> 2,0
0,9 -> 2,9
3,4 -> 1,4
0,0 -> 8,8
5,5 -> 8,2
```

```
import : only (ice-9 rdelim) read-line
         only (srfi :26) cut
         only (srfi :1) fold
         ice-9 hash-table

define : pixels-for-line x0 y0 x1 y1
  cond ;; only vertical and orthogonal lines
    {y0 = y1}
```

```
      map (cut cons <> y0)
        if {x0 < x1} : iota (+ 1 {x1 - x0}) x0
                       iota (+ 1 {x0 - x1}) x1
    {x0 = x1}
      map (cut cons x0 <>)
        if {y0 < y1} : iota (+ 1 {y1 - y0}) y0
                       iota (+ 1 {y0 - y1}) y1
    else '()

define : line-coordinates line
  map string->number : string-tokenize line char-set:digit

define : hash-add1 key al
  hash-set! al key : + 1 : hash-ref al key 0
  . al

define port : open-input-file "advent-of-wisp-code-2021-d5p1-example-input.dot"

display
  hash-count : λ(key value) {value >= 2}
      let loop : : coordinates : make-hash-table
        define line : read-line port
        if : eof-object? line
          . coordinates
          loop
            fold hash-add1 coordinates
                 apply pixels-for-line : line-coordinates line
```

Day 5, Puzzle 2: Sidestep the vents diagonally

Draw lines and find meeting points.

```
0,9 -> 5,9
8,0 -> 0,8
9,4 -> 3,4
2,2 -> 2,1
7,0 -> 7,4
6,4 -> 2,0
0,9 -> 2,9
3,4 -> 1,4
0,0 -> 8,8
5,5 -> 8,2
```

```
import : only (ice-9 rdelim) read-line
         only (srfi :26) cut
         only (srfi :1) fold
         ice-9 hash-table

define : pixels-for-line x0 y0 x1 y1
  cond ;; only vertical and orthogonal lines
    {y0 = y1}
      map (cut cons <> y0)
        if {x0 < x1} : iota (+ 1 {x1 - x0}) x0
                       iota (+ 1 {x0 - x1}) x1
    {x0 = x1}
      map (cut cons x0 <>)
        if {y0 < y1} : iota (+ 1 {y1 - y0}) y0
                       iota (+ 1 {y0 - y1}) y1
    else
      map cons
        if {x0 < x1} : iota (+ 1 {x1 - x0}) x0
                       iota (+ 1 {x0 - x1}) x0 -1
        if {y0 < y1} : iota (+ 1 {y1 - y0}) y0
                       iota (+ 1 {y0 - y1}) y0 -1


define : line-coordinates line
  map string->number : string-tokenize line char-set:digit

define : hash-add1 key al
  hash-set! al key : + 1 : hash-ref al key 0
  . al

define port : open-input-file "advent-of-wisp-code-2021-d5p1-example-input.dot"

display
  hash-count : λ(key value) {value >= 2}
      let loop : : coordinates : make-hash-table
        define line : read-line port
        if : eof-object? line
          . coordinates
          loop
            fold hash-add1 coordinates
                 apply pixels-for-line : line-coordinates line
```

Day 6, Puzzle 1: Model Exponential Fish

Strange lanternfishes reproduce every 7 days, new fish initially reproduce after 9 days. Model the population growth.

How many will there be after 80 days?

Input: The time to reproduce for each fish.

```
3,4,3,1,2
```

```
import : only (srfi :1) fold
         only (ice-9 rdelim) read-line

define input
  let : : port : open-input-file "advent-of-wisp-code-2021-d6p1-example-input.dat"
    define res : map string->number : string-split (read-line port) #\,
    close port
    . res

define : reproduce time-to-reproduce prev
  if : zero? time-to-reproduce
       cons 8 : cons 6 prev
       cons {time-to-reproduce - 1} prev

display
 length
  let rep : (steps 80) (swarm input)
    if (zero? steps) swarm
      rep {steps - 1} : fold reproduce '() swarm
```

Day 6, Puzzle 2: Model Exponential Fish in Memory

OK, 256 days. That kills my memory for sure. Need a tighter datastructure. Let's use the keys for the lifetimes. The keys are contiguous integers, so why not a vector?

```
3,4,3,1,2
```

```
import : only (srfi :1) fold
         only (ice-9 rdelim) read-line

define input
  let : : port : open-input-file "advent-of-wisp-code-2021-d6p1-example-input.dat"
    define res : map string->number : string-split (read-line port) #\,
    close port
    . res
```

```
define swarm-lifetime-counts
  let : : swarm : make-vector 9 0
    for-each : λ (x) : vector-set! swarm x : + 1 : vector-ref swarm x
      . input
    . swarm

define : reproduce swarm
  define reproducing : vector-ref swarm 0
  ;; reduce all lifetimes by 1
  for-each
    λ (lifetime)
      vector-set! swarm {lifetime - 1} : vector-ref swarm lifetime
    iota 8 1
  ;; add the reproducing to lifetime 6 and 8
  vector-set! swarm 6 : + reproducing : vector-ref swarm 6
  vector-set! swarm 8 reproducing
  . swarm

display
  apply +
    vector->list
      let rep : (steps 256) (swarm swarm-lifetime-counts)
        if (zero? steps) swarm
          rep {steps - 1} : reproduce swarm
```

Since the fish with the real data are in the trillions, no way I could have done this with the plain list. Each pointer in a linked list needs around 8 byte; just the datastructure would have eaten all my memory many times over. Even a naively optimized tight array with 3-bit-numbers would not have enabled that.

With the new index-counting vector datastructure though, I can easily do 2560 steps. With the example data, the resulting number has 98 digits. 256000 steps take about a second to compute a number with 9687 digits.

Computers are fast.

Day 7, Puzzle 1: Align Fuel Constrained Crab Guns

Crabs come to blast a path into a cave. You must align them: Find the positions where they need to move the least amount of steps so their guns can interlock into one big gun.

```
16,1,2,0,4,2,7,1,2,14
```

```
import : only (ice-9 rdelim) read-line
```

```
         only (srfi :26) cut
         only (srfi :1) list-index list-ref

define crabs
  let : : port : open-input-file "advent-of-wisp-code-2021-d7p1-example-input.dat"
    define line : read-line port
    close port
    map string->number : string-split line #\,

define min-position : apply min crabs
define max-position : apply max crabs

define possible-positions
  iota (+ 1 {max-position - min-position}) min-position

define : fuel-cost target-position crabs
  define : fuel-cost crab
    abs {crab - target-position}
  apply + : map fuel-cost crabs

define costs : map (cut fuel-cost <> crabs) possible-positions
define min-cost : apply min costs
define ideal-position
  list-ref possible-positions
    list-index (cut equal? min-cost <>) costs

display min-cost
```

Day 7, Puzzle 2: Align Stingy Crab Guns

Movement cost now increases by one per step. Step 1 is 1. Step 2 costs 2, so it is 3. Formula: (step * (step + 1)) / 2

```
16,1,2,0,4,2,7,1,2,14
```

```
import : only (ice-9 rdelim) read-line
         only (srfi :26) cut
         only (srfi :1) list-index list-ref

define crabs
  let : : port : open-input-file "advent-of-wisp-code-2021-d7p1-example-input.dat"
    define line : read-line port
    close port
```

```
    map string->number : string-split line #\,

define min-position : apply min crabs
define max-position : apply max crabs

define possible-positions
  iota (+ 1 {max-position - min-position}) min-position

define : fuel-cost target-position crabs
  define : distance crab
    abs {crab - target-position}
  define : cost crab
    define dist : distance crab
    * 1/2 dist {dist + 1}
  apply + : map cost crabs

define costs : map (cut fuel-cost <> crabs) possible-positions
define min-cost : apply min costs
define ideal-position
  list-ref possible-positions
    list-index (cut equal? min-cost <>) costs

display : format #f "position: ~a, cost: ~a" ideal-position min-cost
```

Day 8, Puzzle 1: Which numbers are shown?

I'm late on this, because a brief solution wasn't directly obvious and I didn't have much time.

I have 10 patterns and 4 displays. Four numbers use a unique number of connections:

- 1: 2
- 4: 4
- 7: 3
- 8: 7

So basically I just need to count occurence of length of strings.

Input:

```
be cfbegad cbdgef fgaecd cgeb fdcge agebfd fecdb fabcd edb | fdgacbe cefdb cefbgd gcb
edbfga begcd cbg gc gcadebf fbgde acbgfd abcde gfcbed gfec | fcgedb cgb dgebacf gc
fgaebd cg bdaec gdafb agbcfd gdcbef bgcad gfac gcb cdgabef | cg cg fdcagb cbg
```

```
fbegcd cbd adcefb dageb afcb bc aefdc ecdab fgdeca fcdbega | efabcd cedba gadfec cb
aecbfdg fbg gf bafeg dbefa fcge gcbea fcaegb dgceab fcbdga | gecf egdcabf bgf bfgea
fgeab ca afcebg bdacfeg cfaedg gcfdb baec bfadeg bafgc acf | gebdcfa ecba ca fadegcb
dbcfg fgd bdegcaf fgec aegbdf ecdfab fbedc dacgb gdcebf gf | cefg dcbef fcge gbcadfe
bdfegc cbegaf gecbf dfcage bdacg ed bedf ced adcbefg gebcd | ed bcgafe cdgba cbgef
egadfb cdbfeg cegd fecab cgb gbdefca cg fgcdab egfdb bfceg | gbdfcae bgc cg cgb
gcafb gcf dcaebfg ecagb gf abcdeg gaef cafbge fdbac fegbdc | fgae cfgab fg bagce
```

```
import : only (ice-9 rdelim) read-line
         srfi :9 records
         only (srfi :26) cut
         only (srfi :1) second

define : split-result-into-length line
  map string-length
    string-tokenize
      second : string-split line #\|
      . char-set:letter

define : map-over-lines fun filename
  let : : port : open-input-file filename
    let loop : (lines '()) (line (read-line port))
      if : eof-object? line
        begin
          close port
          reverse! lines
        loop
          append : fun line
                 . lines
          read-line port

define input
  map-over-lines split-result-into-length
    . "advent-of-wisp-code-2021-d8p1-example-input.dat"

define counter : make-vector 8 0

for-each : λ(len) : vector-set! counter len : + 1 : vector-ref counter len
         . input

display
  apply +
    map (cut vector-ref counter <>)
      list 2 4 3 7
```

Day 8, Puzzle 2: Which numbers are shown?

Now do the full mapping.

Use the left-hand patterns to recover the configuration.

```
import : only (ice-9 rdelim) read-line
         srfi :9 records
         only (srfi :26) cut
         only (srfi :1) first second fold assoc
         only (rnrs lists (6)) find

;; the numbers
;;   0:      1:      2:      3:      4:
;;  aaaa    ....    aaaa    aaaa    ....
;; b    c  .    c  .    c  .    c  b    c
;; b    c  .    c  .    c  .    c  b    c
;;  ....    ....    dddd    dddd    dddd
;; e    f  .    f  e    .  .    f  .    f
;; e    f  .    f  e    .  .    f  .    f
;;  gggg    ....    gggg    gggg    ....
;;
;;   5:      6:      7:      8:      9:
;;  aaaa    aaaa    aaaa    aaaa    aaaa
;; b    .  b    .  .    c  b    c  b    c
;; b    .  b    .  .    c  b    c  b    c
;;  dddd    dddd    ....    dddd    dddd
;; .    f  e    f  .    f  e    f  .    f
;; .    f  e    f  .    f  e    f  .    f
;;  gggg    gggg    ....    gggg    gggg

;; define number deciders
define : 1? string
  = 2 : string-length string
define : 4? string
  = 4 : string-length string
define : 7? string
  = 3 : string-length string
define : 8? string
  = 7 : string-length string
define : 2-or-3-or-5? string
  = 5 : string-length string
define : 0-or-6-or-9? string
  = 6 : string-length string
```

```
;; returns (pattern-part result-part)
define : split-into-strings line
  map : cut string-tokenize <> char-set:letter
    string-split line #\|

define : map-over-lines fun filename
  let : : port : open-input-file filename
    let loop : (lines '()) (line (read-line port))
      if : eof-object? line
        begin
          close port
          reverse! lines
        loop
          cons : fun line
               . lines
          read-line port

define input-strings
  map-over-lines split-into-strings
    . "advent-of-wisp-code-2021-d8p1-real-input.dat"

define input-charsets
  map
    λ(line)
      map : λ(x) : map string->char-set x
          . line
    . input-strings

define : process-one-line line-strings line-charsets
  ;; identify the char-sets for digits of unique length
  define pattern-strings
    first line-strings
  define result-charsets
    second line-charsets
  define : find-matching-charsets string-matches? pattern-strings
           fold : λ(string prev) : append (if (string-matches? string) (list (string-
                . '() pattern-strings
  define one : first : find-matching-charsets 1? pattern-strings
  define four : first : find-matching-charsets 4? pattern-strings
  define seven : first : find-matching-charsets 7? pattern-strings
  define eight : first : find-matching-charsets 8? pattern-strings
  define zero-or-six-or-nine
    find-matching-charsets 0-or-6-or-9? pattern-strings
```

```
define six
  find : λ(x) : not : char-set<= one x
       . zero-or-six-or-nine
define zero-or-nine
  filter : λ(x) : char-set<= one x
       . zero-or-six-or-nine
define nine
  find : λ(x) : char-set<= four x
    . zero-or-nine
define zero
  find : λ(x) : not : char-set<= four x
    . zero-or-nine
define two-or-three-or-five
  find-matching-charsets 2-or-3-or-5? pattern-strings
define three
  find : λ(x) : char-set<= one x
       . two-or-three-or-five
define five
  find : λ(x) : char-set<= x nine
       delete three two-or-three-or-five
define two
  first : delete five : delete three two-or-three-or-five

define known-charsets
  list
    cons 1 one
    cons 4 four
    cons 7 seven
    cons 8 eight
    cons 6 six
    cons 9 nine
    cons 0 zero
    cons 3 three
    cons 5 five
    cons 2 two
define charset-to-number
      map : λ(x) : cons (cdr x) (car x)
          . known-charsets

string->number
  string-join
    map number->string
      map : λ(x) : cdr : assoc x charset-to-number char-set=
          . result-charsets
```

```
      . ""

write : apply + : map process-one-line input-strings input-charsets
```

This one was long, far longer than I would have liked. And with much more logic coming in from me instead of the program. I wonder if micro-/minikanren or Prolog would provide for a nicer solution.

Day 9, Puzzle 1: Avoid smoke-sinks

Find low points in height-map.

```
2199943210
3987894921
9856789892
8767896789
9899965678
```

```
import : only (ice-9 rdelim) read-line
         only (ice-9 pretty-print) pretty-print
         only (srfi :1) fold

define input
  let : : port : open-input-file "advent-of-wisp-code-2021-d9p1-example-input.dat"
    let loop : (lines '()) (line (read-line port))
      if : eof-object? line
         list->vector : reverse! lines
         loop
           cons : list->vector : map string->number : map string : string->list line
                . lines
           read-line port

define max-y : 1- : vector-length input
define max-x : 1- : vector-length : vector-ref input 0

define : at vec x y
  vector-ref (vector-ref input y) x

define : low-point? input x y
  define up : and {y > 0} {y - 1}
  define down : and {y < max-y} {y + 1}
  define left : and {x > 0} {x - 1}
  define right : and {x < max-x} {x + 1}
  define val : at input x y
```

```
  and : if up (< val (at input x up)) #t
        if down (< val (at input x down)) #t
        if left (< val (at input left y)) #t
        if right (< val (at input right y)) #t


define risk-levels
 map
  λ(y)
    map : λ(x) : if (low-point? input x y) (+ 1 (at input x y)) 0
              iota : + 1 max-x
  iota : + 1 max-y

pretty-print : apply + : map (λ(row) (apply + row)) risk-levels
```