

MAU11601

Lecture 06

2026-10-06

Names

You need names for a lot of things. Racket does not care what you name things, as long as you're consistent. People who read your code (including you) probably do. Conventions are good. Some are standard, e.g. `?` for predicates. Some you need to create yourself, e.g. "in this programme `p` and `q` are always `posns` and `ps` or `qs` are always lists of `posns`."

Abbreviate consistently or not at all.

Long descriptive names are good for things which appear far apart in the code, e.g. names of most functions, fields of a structure, etc.

Short names are good for things that only live in the reader's short term memory, e.g. arguments to short functions, names of auxiliary functions used only in the function immediately following, etc.

I am bad at naming things. Don't take my code as a model.

AoC 2025 Day 1, again

Last time we did Part 1 of Advent of Code 2025 Day 1 with an interactive graphical programme.

That's nice as an illustration of how to write interactive graphical programmes, but overkill for the actual programme.

A more standard way to do it is as a batch programme.

- Read the whole input all at once.
- Split it into a list of strings, one for each line.
- Process the list using `foldl`, with the first argument a function which updates the state based on one line.
- Read the answer off from the final state.

In the process we'll learn about some Racket string functions.

Fold and big-bang

`foldl` is a bit like `big-bang`. It applies a user supplied function repeatedly, starting from a user supplied initial state, using extra information from somewhere, returning the final state.

- For `big-bang` the extra information comes from events, e.g. clock ticks, keyboard presses, mouse movements, etc.
- For `foldl` it comes from a list.

There are additional differences. `big-bang` also takes a user supplied function to draw scenes as it goes, while `foldl` doesn't.

Also, `foldl` only takes one user supplied function to say what to do with list elements. `big-bang` has separate functions for different types of events.

That's purely for convenience. It could take one event handler, with a function to identify event type and a `big cond` statement to choose the appropriate action, like we already do within event types for keys and mouse events.

The similarities are more important than the differences though. We should be able to convert `big-bang` program to a `foldl` program and vice versa.

Creating our list

For the `foldl` version of our AoC we need a list, and the obvious thing to use is a list of dial movements, represented as integers, positive or negative, depending on direction.

To get it, we read the input, split it into lines, and convert each line to a number.

`read-file` opens a file, reads it and returns a string with its contents, closing the file when done. It's in the `2htpd/batch-io` module so we'll need to `require` that.

```
(define (split-into-lines s) (string-split s "\n"))
```

splits a string into a list of strings, one per line.

`(substring s 0 1)` returns a substring with just the first character and

`(substring s 1)` returns the rest.

```
(define (convert-line s)
```

```
  (* (if (equal? (substring s 0 1) "L") -1 1)
```

```
      (string->number (substring s 1))))
```

converts a line of input into the corresponding integer.

Creating our list, continued

We now have all the pieces so we just need to put them together.

If we've copied the example data

L68

L30

R48

L5

R60

L55

L1

L99

R14

L82

from the problem statement into the file `example1.txt` then

```
(map convert-line (split-into-lines (read-file "example1.txt")))
```

will give us the list of integers `(list -68 -30 48 -5 60 -55 -1 -99 14 -82)`.

State

For the interactive programme we had a `world` structure with five fields, `points-to`, `count`, `direction`, `clicks-left` and `nsf`. Which of these do we need for our batch version?

The first two are still needed. We need to know the current dial position and we need to keep track of the zeroes we've seen.

We don't need `direction`. That information is encoded in the sign of our list elements. We don't need `clicks-left`, which was used to animate the dial movement. We don't need `nsf`, which was used to convert a sequence of digit keys into a number.

So the following should suffice.

```
(define-struct state (points-to count))  
(define start (make-state 50 0))
```

We now have the initial state and the list for `foldl`. All that's left is the function to update the state.

The update function and final programme

The update function just needs to add the list element to the `points-to` field and then increment the `count` field if needed.

```
(define (new-points-to e s)
  (modulo (+ e (state-points-to s)) 100))
(define (new-count e s)
  (+ (if (zero? (new-points-to e s)) 1 0) (state-count s)))
(define (new-state e s)
  (make-state (new-points-to e s) (new-count e s)))
```

I defined auxiliary functions `new-points-to` and `new-count` for the individual fields. I could have just stuck everything into `new-state` instead.

Now `(foldl new-state start data)` will return the final state, but we only care about its `count` field so the expression we need to evaluate to solve the problem is

```
(state-count (foldl new-state start data))
```

For what it's worth, the batch solver is 83% shorter than the interactive one.

Part II

As I mentioned, each Advent of Code puzzle has two parts, with the second unlocked only after you've solved the first.

Usually if you've thought about the first part in the right way the second part is an easy modification.

In this case the second part is the same, except we count all the times the arm points to 0, not just the times when it does so after a rotation.

For our interactive programme this is very easy. We tracked all the numbers it pointed to and incremented count when `points-to` and `ticks-left` were both 0. For the second part we can just remove the check on `ticks-left`.

The batch programme is a bit harder to modify. The easiest way to do it is add another stage in the list processing, converting one -68 into 68 -1's, one 48 into 48 1's, etc. It's not too hard to do this with some folds, but I won't.

Mixed programmes

HtDP treats interactive and batch programmes as a binary choice, and that's technically correct, but many programmes look like hybrids.

Consider our unfinished maze programme. We can generate the maze, which is hard, or read one or more prepared ones from a file. A natural file format is something like

```
#####  
S..#...#E#  
##.#.#.#.#  
#....###.#  
#.#.#.....#  
#.#####  
#.....#  
#####
```

Your programme might have two parts, one in batch style which reads the file and constructs the maze, and one in interactive style, where the user navigates the maze.

Debugging

Almost no one likes debugging, but there are things you can do to make it less terrible.

DrRacket has a stepper you can use to see how expressions get evaluated!

Try to get an error as soon as things go wrong. `big-bang` has a `check-with` parameter for a predicate which performs sanity checks on the world state.

When you write a function, test it with some sample arguments in the interaction window.

It's easier to test a lot of simple functions than one complicated one, so split your task into small pieces.

Beware of trivial cases! List functions, for example, are more likely to fail on an empty list than a non-empty one.

Start from a trivial programme which works, then make it non-trivial. Don't start from a non-trivial programme which doesn't work. There was a reason my first attempt at a maze programme had no walls.

This is especially important for your group project. Your programme must work, even if it doesn't do everything you wanted!

Don't trust your own comments!