

MAU11601

Lecture 07

2026-10-07

More list functions

There are a bunch of useful functions for manipulating lists.

`length`, `append` and `reverse` all do what you would expect.

`sort` also does what you'd expect, except you have to give it two arguments, the list and a function to determine the order of pairs of elements. To sort a numerical list `l` in ascending order you would use `(sort l <)`.

`empty?` is a predicate which checks whether the list is empty, as you would expect.

`null?` is an older, but less informative name for this function.

`first` and `rest` give you the first element of a non-empty list and a list containing the rest of the elements, but give you an error if the list is empty. `car` and `cdr` are older, less informative, names for them.

There's also a `last`, but it's much slower than `first`, because of how lists are represented.

You should only ever use `first` and `rest` inside a conditional where you check `empty?` first and have a branch saying what to do if the list is empty.

Recursive functions, map

Some things which look like functions aren't really, like `and` and `or`.

Other examples we've met already are `if`, `cond`, `define` and `define-struct`.

The big one is `big-bang`.

By contrast, `map`, `filter`, `foldl` and `foldr` are ordinary functions.

They happen to come predefined, but if they didn't you could define them yourself.

```
(define (_map f l)
  (if (empty? l) empty (cons (f (first l)) (_map f (rest l)))))
```

I called it `_map` rather than `map` to avoid confusion with the built-in `map` function.

Note that this function is recursive. It calls itself in the `else` branch of the conditional.

In general recursive functions may fail to terminate, like

```
(define (loop x) (loop x)).
```

The reason `_map` is okay is that each time it's called it's called with a list of length one less, so eventually it's called on the empty list and returns without calling itself again.

Since this is a maths module we can say that for all natural numbers n the function `_map` applied to a list of length n terminates, and this is proved by induction on n .

Filter and the folds

```
(define (_filter p? l)
  (if (empty? l)
      empty
      (if (p? (first l))
          (cons (first l) (filter p? (rest l)))
          (_filter p? (rest l)))))
```

This terminates for the same reason `filter` does.

```
(define (_foldl f s l)
  (if (empty? l) s (_foldl f (f (first l) s) (rest l))))
(define (_foldr f s l)
  (if (empty? l) s (f (first l) (_foldr f s (rest l)))))
```

If you want to understand how these things work DrRacket's stepper is your friend.

Tail recursion

`_map`, `_filter`, `_foldl`, `_foldr` and our ill-fated loop were all defined recursively, but not all recursions are the same.

```
(define (reject n)
  (error "expected nonnegative integer but got " n))

(define (f n)
  (cond [(zero? n) 1]
        [(positive? n) (* n (f (- n 1)))]
        [else (reject n)]))

(define (g n a)
  (if (zero? n)
      a
      (g (- n 1) (* n a))))

(define (h n)
  (cond [(zero? n) 1]
        [(positive? n) (g n 1)]
        [else (reject n)]))
```

Tail recursion and folds

```
(define (_foldl f s l)
  (if (empty? l) s (_foldl f (f (first l) s) (rest l))))
(define (_foldr f s l)
  (if (empty? l) s (f (first l) (_foldr f s (rest l)))))
```

These are recursive, i.e. they call themselves, but they do it in slightly different ways.

`(_foldr + 0 '(1 1 2 3 5 8 13))` has intermediate steps like
`(+ 1 (+ 1 (+ 2 (+ 3 (_foldr + 0 '(5 8 13))`. In other words, we're building up a sequence of operations to perform and then performing them all at the end. `_foldl` doesn't do this. It's intermediate steps are all just calls to `_foldl`, like `(_foldl + 8 '(5 8 13))`.

This is because `_foldl` doesn't perform any work after the recursive call. This kind of recursion is called tail recursion, and it makes `_foldl` more efficient than `_foldr`.

The definitions of `_map` and `_filter` given earlier are not tail recursive, but it is possible to write tail recursive implementations.

To roll your own, or not?

Recursion has a reputation for difficulty, but it's not really that hard. You can write your own recursive functions.

You can, but should you? Mostly no. If you can use maps, filters and folds instead then you generally should.

- Definitions in terms of maps, filters, folds, etc. are usually clearer than recursive definitions.
- It localises the burden of proving termination. You prove once and for all that those functions terminate when given a terminating function as an argument.
- The standard versions are designed for efficiency. Your hand coded version is likely to be less efficient, in time or space or both.

AoC 2025 Day 5

You want to read the input and split it into two lists, one of ranges and one of values. We can use `string-split` for this.

We'll certainly need a function to test whether a value lies in a range.

```
(define (in-range? v r) (and (>= v (first r)) (<= v (second r))))
```

If we've stored our values in `vs` and written a predicate `in-some-range?` to test whether a value lies in at least one of our ranges, the values in some range will be `(filter in-some-range? vs)` and we can get the number of them using `length`. How do we write `in-some-range?`?

This should be easy with `ormap` and `in-range?`, but there's a problem. The predicate we give `ormap` needs to be a function of the range, but `in-range?` has both a value and a range argument. We need to fix the value argument to be the one we got as an argument to `in-some-range?`.

Specialising a function by fixing some of its arguments is called partial evaluation. In Racket it can be done with the `curry` function.

```
(define (in-some-range? v) (ormap (curry in-range? v) rs))
```

Partial evaluation (currying)

When we define a function we're doing two things, creating a function and giving it a name. Often we only need a function once, so it doesn't really need a name.

This is especially true for the functions in maps, filters and folds.

`lambda` is the way we do this in Racket. It's not part of BSL or ISL but it is part of ISL+

Syntactically it looks like an ordinary function definition except we omit the name.

In the expression defining the function we can use any variable which is in scope, i.e. which will have a value in this context.

That includes defined values (including functions) and also arguments of the function within which we are defining our anonymous function.

In the case of `(lambda (r) (in-range? v r))` that would be the `v`.

Really, this isn't a single function but a whole family of functions, parameterised by `v`.

Each time we call `in-some-range?` with a new value of `v` we get a new function.

Racket allows us to create new functions dynamically like this. Not all languages do.