Module 5: Deconstructing and constructing lists

If you have not already, please

- Read the Wikipedia entry on *Filter (higher order function)*.

Working on just part of a list

Problem: Calculate the sum of all multiples of 2, 3, or 5, between 0 and 1000.

Maybe try something like:

```
(define 2-multiples (range 0 1000 2))
(define 3-multiples (range 0 1000 3))
(define 5-multiples (range 0 1000 5))
; ???
```

I can't simply add these up; numbers like 6 would be counted twice, and numbers like 60 would be counted three times.

Perhaps I could do something with `foldr` and `cond`, but it may be tricky. What to do?

Working on just part of a list

I can check a single number easily enough.
The function `muliple-235?` returns **`#true`** if a `Nat` is of the numbers I need to add up:

```
(define (divisible? n d) (= 0 (remainder n d)))

;; (multiple-235? n) return #true if n is divible by 2,3, or 5.
;; multiple-235?: Nat -> Bool

(define (multiple-235? n)
  (or (divisible? n 2) (divisible? n 3) (divisible? n 5)))
```

Somehow I need to keep only these numbers, and add them up.

Another higher order function: `filter`

The built in function **`filter`** does exactly what we need.

> `(filter F L)` consumes a predicate function and a `(listof Any)`. `F` must be a one-parameter `Function` that consumes the type(s) of value in `L`, and returns a `Bool`.
> `(filter F L)` will return a list containing all the items `x` in `L` for which `(F x)` returns **`#true`**.

`(filter F (list x0 x1 x2 ... xn))` $\rightarrow$ `(list x0 x3 ... )`
For all values `xk` for which `(F xk) => #true`.

Another higher order function: filter

For example:

```
;; (keep-multiples-235 L) keep all values in L that are divisible by 2, 3, or 5.
;; keep-multiples-235: (listof Nat) -> (listof Nat)
;; Example:
(check-expect (keep-multiples-235 (range 0 10 1)) (list 0 2 3 4 5 6 8 9))
(check-expect (keep-multiples-235 (list 2 4 7 7 50 4)) (list 2 4 50 4))

(define (keep-multiples-235 L) (filter multiple-235? L))
```

Another example using the built in predicate **even?**:

```
;; keep-even L: keep all even values in L.
;; keep-even: (listof Int) -> (listof Int)
;; Example:
(check-expect (keep-even (list 1 2 3 4 5 6)) (list 2 4 6))
(define (keep-even L) (filter even? L))
```

filter practice

Here is an example of a function using **filter**: $\longrightarrow$

```
(define (not-apple x) (not (equal? x "apple")))
(define (drop-apples L) (filter not-apple L))
```

Exercise

Use **filter** to write a function that consumes a `(listof Num)` and keeps only values between 10 and 30, inclusive.

```
(keep-inrange (list -5 10.1 12 7 30 3 19 6.5 42)) => (list 10.1 12 30 19)
```

Exercise

Use **filter** to write a function that consumes a `(listof Str)` and removes all strings of length greater than 6.

```
;; (keep-short L) Keep all the values in L of length at most 6.
;; keep-short: (listof Str) -> (listof Str)
;; Example:
(check-expect (keep-short (list "Strive" "not" "to" "be" "a" "success"
                                "but" "rather" "to" "be" "of" "value"))
              (list "Strive" "not" "to" "be" "a"
                    "but" "rather" "to" "be" "of" "value"))
```

In combination, these functions are very powerful.

Exercise

Write a function `times-square` that consumes a `(listof Nat)` and returns the product of all the perfect squares $(1, 4, 9, 16, 25, \ldots)$ in the list.

```
(check-expect (times-square (list 1 25 5 4 1 17)) 100)
;; Since (times-square (list 1 25 5 4 1 7)) => (* 1 25 4 1) => 100
```

Hint

Use `integer?` to check if a value is an integer.

```
(integer? (sqrt 5)) => #false
(integer? (sqrt 4)) => #true
```

Constructing lists

Two functions which operate on lists, and which we will use more later, are `first` and `rest`:

```
(define L (list 2 3 5 7 11))
 (first L)       (rest L)
     ↓                ↓
     2        (list 3 5 7 11)
```

`first` consumes a `(listof Any)`, and returns the first value in that list.
`rest` consumes a `(listof Any)`, and returns the list with all the values **except** the first.

We want to go the other way:

We may use cons to construct lists:

- It consumes two values: an Any, and a (listof Any).
- It returns a list one longer, with the new value added at the **left** of the list.

```
(cons 4 (list 1 2 3)) => (list 4 1 2 3)
(cons 1 (cons 2 (cons 3 '()))) => (list 1 2 3)
```

(It's a little trickier to add to the right of a list, or to get the last item.)

Ex. Construct `(list 6 7 42)` using only `cons` and the empty list, `'()`.

Exercise Write a function `remove-second` that consumes a list of length at least two, and returns the same list with the second item removed.

```
(check-expect (remove-second (list 2 4 6 0 1)) (list 2 6 0 1))
```

Using foldr to construct lists

Recall what **foldr** does:

```
(foldr F base (list x0 x1 ... xn)) => (F x0 (F x1 (F ... (F xn base))))
```

We can use **foldr** to copy a list.

```
(foldr cons '() (list 2 3 5))
=> (cons 2 (cons 3 (cons 5 '())))
=> (list 2 3 5)
```

Faking map

We can create new lists using **cons** and **foldr**, as if we were using **map**.

Using **map**, I can add 2 to each value in a list:

```
;; (add-2 x) add 2 to x.
;; add-2: Num -> Num
(define (add-2 x)
  (+ x 2))
```

```
;; (add-2-each-m L) add 2 to each of L.
;; add-2-each-m: (listof Num) ->
     (listof Num)
(define (add-2-each-m L)
  (map add-2 L))
```

I can do the same thing with **foldr** instead:

```
;; (add-2-first a b) contruct a list
;;    using 2 more than a, then b.
;; add-2-first: Num (listof Num) ->
     (listof Num)
(define (add-2-first a b)
  (cons (+ 2 a) b))
```

```
;; (add-2-each-f L) add 2 to each of L.
(define (add-2-each-f L)
  (foldr add-2-first '() L))
```

Ex. Write more tests to verify that `add-2-each-m` and `add-2-each-f` both work.

Exercise

The following function works. Rewrite it using `foldr`, without using `map`.

```
(define (double n) (* n 2))

(define (double-each L) (map double L))
```

```
(foldr F base (list x0 x1 ... xn)) => (F x0 (F x1 (F ... (F xn base))))
```

Faking filter using foldr

We can create new lists using **cons** and **foldr**, as if we were using **filter**.

Using **filter**, I can keep items bigger than 5:

```
;; (big? x) Is x > 5?
;; big?: Num -> Bool
(define (big? x)
  (> x 5))


;; (keep-big L) keep big vals from L.
;; keep-big: (listof Num) ->
       (listof Num)
(define (keep-big L)
  (filter big? L))
```

I can do the same thing with **foldr** instead:

```
;; (add-if-big item oldL)
;;   Add big item before oldL;
;;     otherwise ignore it.

(define (add-if-big item oldL)
  (cond [(big? item) (cons item oldL)]
        [else oldL]))


;; (keep-big-f L) keep big vals from L.
;; keep-big-f: (listof Num) ->
     (listof Num)
(define (keep-big-f L)
  (foldr add-if-big '() L))
```

Ex. Write more tests to verify that `keep-big` and `keep-big-f` both work.

Faking filter using foldr

```
(foldr F base (list x0 x1 ... xn)) => (F x0 (F x1 (F ... (F xn base))))
```

Exercise

Using `foldr`, write a function `(keep-evens L)` that returns the list containing all the even values in `L`.

That is, rewrite this function, using `foldr` but not using `filter`:

```
(define (keep-evens L)
  (filter even? L))

(check-expect (keep-evens (list 1 2 3 4 5 6)) (list 2 4 6))
```

Hint

With `foldr` you have the "partial answer" from the previous call, which here must be a `(listof Int)`.

- Sometimes, you want to `cons` the new value to the old answer.
- Sometimes you want to ignore the new value, and just return the old answer.

Overview of Higher Order Functions

map Transforms each item in a list, and returns a list of the **same size**.

`(map F (list x0 x1 ... xn)) => (list (F x0) (F x1) ... (F xn))`

`(map sqr (list 2 3 5)) => (list 4 9 25)`

filter Consider each item in a list, and returns a **list of the same items** for which the predicate returns `#true`. This list will usually be smaller.

`(filter G (list x0 x1 ... xn)) => (list x0 x2)`, if `x0` and `x2` are the only values in the list for which `G` returns `#true`.

`(filter even? (list 2 5 8 7 4 3 2)) => (list 2 8 4 2)`

foldr Combine items in a list, and return **a single value**.
This could be of any type, even a list.

`(foldr H base (list x0 x1 ... xn)) => (H x0 (H x1 (H ... (H xn base))))`

`(foldr * 7 (list 2 10 3)) => 420`

If your function consumes a list, you may want to use one or more higher order functions. How to decide which one to use? Consider your **desired output**.

desired output	likely solution
a list the same size as the input	consider `map`
a list containing some of the items from the input	consider `filter`
a single value	consider `foldr`
a list, but not something you can do with `map` and `filter`	consider `foldr`, using `cons`

You may prefer to use some combination of these functions.

Data-driven design: some hints on how to use `foldr`

Recall what **`foldr`** does:

`(foldr F base (list x0 x1 ... xn)) => (F x0 (F x1 (F ... (F xn base))))`

What does this tell us about the contract for `(F a b)` ?

1. It says `(F x0 ...)` , `(F x1 ...)`, etc.
   So the first parameter has to be the same as the type of the values in the list.
2. It says `(F ... (F ...))`.
   So whatever value `F` returns will be used as the second parameter of `F`.
   So the return value and the second parameter must be of the same type.
3. It says `(F ... base)`, so the `base` is also of this type.

That is, to write `(foldr F base L)`, with `L` a `(listof X)`, the contract for `F` must be of the form:

```
F: X Y -> Y
```

...and `base` must be of type `Y`.

Exercise

Given that `use-foldr` consumes a `(listof Nat)`:

```
(define (use-foldr L) (foldr myfun "some-str" L))
```

carefully consider:

1. What is the contract for `myfun` ?
2. What is the contract for `use-foldr` ?

Ex. Write a function `myfun` that allows `use-foldr` to do something.

Consider this function:

```
(define (myfun n s) (string-append (number->string n) s))
(foldr myfun base L)
```

What can we say about `base` and `L`?

- `n` must be a `Num`, so `L` must be a `(listof Num)`.
- `s` must be a `Str`, so `base` number be a `Str`.
- It is good that `myfun` returns a value of the same type as `s`.

We haven't yet seen the whole power of **foldr**.

Consider: if I have a `(listof Num)`, I want to be able to find the largest value in the list. For example, the largest value in `(list 2 -59 42 6 27)` is `42`.

I can use **foldr** to get the largest value, something like this:

```
(define (list-max L)
  (foldr F base L)
  )
```

Exercise

- What is the contract for `list-max`?
- What is type of `base`?
- What is the contract for `F`?

More with `foldr`

Since the final answer is a `Num`,

- We have
  ```
  ;; list-max: (listof Num) -> Num
  ```
- `base` must be a `Num`.
- We have
  ```
  ;; F: Num Num -> Num
  ```

`base` needs to be some `Num`. We need to think about what `Num`, but for now, just use 0.

More with `foldr`

Let's use some helpful variable names in defining `F`. We have:

`(F new-item largest-so-far)` consumes two `Num`.

- `new-item` is an item from the list.
- `largest-so-far` is the largest item we have found so far.

Two examples to consider:

- What should we return if `largest-so-far` is 27, and `new-item` is 6?
- What should we return if `largest-so-far` is 27, and `new-item` is 42?

Exercise

Replace `base` with `0`.

Write `F` so `list-max` works, at least for some inputs.

```
(check-expect (list-max (list 2 4 6 0 1)) 6)
(check-expect (list-max (list 2 -59 42 6 27) 42)
```

More with foldr

Exercise

You may have a bug in your code. Try out the following test:

```
(check-expect (list-max (list -3 -17 -5)) -3)
```

Then change `list-max` so it passes this test.

A new command in a new language level

At this point we introduce a new command, **`lambda`**, which is not a part of the language we have used so far.

> ! In Racket, select *Language* → *Choose language* → *Intermediate student with lambda*.

We will stay in this new language level for the remainder of the term.

Some simple things are annoying

If I wanted to, for example, double each item in a list:

```
;; (double n) return 2*n.
;; double: Num -> Num
;; Examples:
(check-expect (double 3) 6)
(check-expect (double 0) 0)

(define (double n) (* n 2))

;; (double-each L) return L, with each value doubled.
;; double-each: (listof Num) -> (listof Num)
;; Examples:
(check-expect (double-each '()) '())
(check-expect (double-each (list 2 3 5)) (list 4 6 10))

(define (double-each L) (map double L))
```

Half the work is the design recipe for a really simple function!

For short functions which are used just once, `lambda` lets us write **anonymous functions**.

An example:

```
;; (double-each2 L) return L, with each value doubled.
;; double-each2: (listof Num) -> (listof Num)
;; Examples:
(check-expect (double-each2 '()) '())
(check-expect (double-each2 (list 2 3 5)) (list 4 6 10))

(define (double-each2 L)
  (map (lambda (n) (* n 2))
       L))
```

Remember: we use `map` like: `(map Function List)`.

Here `(lambda (n) (* n 2))` takes the place of the `Function`.

That `lambda` expression **is** a `Function`.

Tiny Functions with lambda

lambda is a special form that returns a function.

`(lambda (x) (+ x 7))` is a function with one parameter.

`(map (lambda (x) (+ x 7)) (list 2 3 5)) => (list 9 10 12)`

Exercise

Using **lambda** and **map**, but no [named] helper functions, write a function `cube-each` that consumes a `(listof Num)` and returns a list containing the cube of each `Num`. (x^3)

Exercise

Using `lambda` and `filter` but no named helper functions, write a function that consumes a `(listof Str)` and returns a list containing all the strings that have a length of 4.

```
(keep4 (list "There's" "no" "fate" "but" "what" "we" "make" "for" "ourselves"))
=> (list "fate" "what" "make")
```

Exercise

Using `lambda` but no named help functions, write a function that consumes a `(listof Int)` and returns the sum of all the even values.

```
(sum-evens (list 2 3 4 5)) => 6
```

Can you do it using `lambda` just once and `foldr` just once?

Suppose I wanted to add 5 to every item in a list:

```
;; (add-5 n) add 5 to n.
;; add-5: Num -> Num
(define (add-5 n) (+ n 5))

;; (add-5-each L) add 5 to each item in L.
;; add-5-each: (listof Num) -> (listof Num)
(define (add-5-each L) (map add-5 L))

(check-expect (add-5-each (list 3.2 6 8)) (list 8.2 11 13))
```

This works!

But now suppose I want to be able to add some other value to each. There's a problem: if I add a parameter `n` to `add-5-each`, there is no way for that value to be available to `add-5`.

Handling extra parameters with lambda

We can fix it using **lambda**!

```
;; (add-n-each L n) add n to each item in L.
;; add-n-each: (listof Num) Num -> (listof Num)
(define (add-n-each L n)
  (map (lambda (x) (+ x n))
       L))

(check-expect (add-n-each (list 3.2 6 8) 6) (list 9.2 12 14))
```

This **lambda** function, since it is inside `add-n-each`, can use the value of `n`.

Exercise

Write a function `(multiply-each L n)`. It consumes a `(listof Num)` and a `Num`, and returns the list containing all the values in `L`, each multiplied by `n`.

`(multiply-each (list 2 3 5) 4) => (list 8 12 20)`

Exercise

Write a function `(add-total L)` that consumes a `(listof Num)`, and adds the total of the values in `L` to each value in `L`.

`(add-total (list 2 3 5 10)) => (list 22 23 25 30)`

Using **lambda** we can create a constant which stores a function.

```
(define double (lambda (x) (* 2 x)))
(double 5) => 10
```

(If you do this, you are creating a named function, so you must use the design recipe!)

You can use a **lambda** expression anywhere you need a function:

```
((lambda (x y) (+ x y y)) 2 5) => 12
```

Anything that can go in a function can go in a **lambda**, even another **lambda**:

```
((lambda (x y)
  ((lambda (z) (+ x z)) y)) 4 5)
```

Earlier we had the following functions:

```
(define (divisible? n d) (= 0 (remainder n d)))

(define (multiple-235? n)
  (or (divisible? n 2) (divisible? n 3) (divisible? n 5)))

(define (keep-multiples-235 L) (filter multiple-235? L))
```

Suppose I wanted to keep multiples of a `Nat` which is a parameter:

```
;; (keep-multiples d L) return all values in L which are divisible by d.
;; keep-multiples: Nat (listof Nat) -> (listof Nat)
;; Examples:
(check-expect (keep-multiples 7 (list 2 3 5 28 7 3 14 77)) (list 28 7 14 77))
```

I would like to use **`filter`**, but recall: the `Function` it consumes must have only one parameter. The function `divisible?` has two parameters, `n` and `d`. How can I tell it the `d`?

Solution: use **lambda**.

```
;; (keep-multiples d L) return all values in L which are divisible by d.
;; keep-multiples: Nat (listof Nat) -> (listof Nat)
;; Examples:
(check-expect (keep-multiples 7 (list 2 3 5 28 7 3 14 77)) (list 28 7 14 77))

(define (keep-multiples d L)
  (filter (lambda (v) (divisible? v d)) L))
```

The `n` and `L` variables are **in scope** inside the **lambda** function. It can use them!

Exercise

Write `(discard-bad L lo hi)`. It consumes a `(listof Num)` and two `Num`. It returns the list of all values in `L` that are between `lo` and `hi`, inclusive.

```
(discard-bad (list 12 5 20 2 10 22) 10 20) => (list 12 20 10)
```

Exercise

Write `(squash-bad lo hi L)`. It consumes two `Num` and a `(listof Num)`. Values in `L` that are greater that `hi` become `hi`; less that `lo` become `lo`.

```
(squash-bad 10 20 (list 12 5 20 2 10 22))) => (list 12 10 20 10 10 20)
```

Exercise

Write a function `above-average` that consumes a `(listof Num)` and returns the list containing just the values which are greater than or equal to the average (mean) value in the list.

Hint

You can compute the mean as follows:

```
(define (mean L) (/ (foldr + 0 L) (length L)))
```

Using `map` with `range` we can only create a single list. How to create a list that contains lists?

Idea: write a function that uses `map` to create *one row* of the table.
Then use this function inside another call to `map`.

Times Table

We want to be able to make a times table, something like the following:

```
(timestable 5) =>
  (list (list  1  2  3  4  5)
        (list  2  4  6  8 10)
        (list  3  6  9 12 15)
        (list  4  8 12 16 20)
        (list  5 10 15 20 25))
```

The first step is to write a helper function that creates one row of the table.

Exercise

Write a function `(times-row n len)` that returns the `n`th row of the times table. This should be a list of length `len`. Write you function in the form `(map ... (range 1 (+ len 1) 1))`.

```
(check-expect (times-row 3 5) (list 3 6 9 12 15))
(check-expect (times-row 6 3) (list 6 12 18))
```

Hint

Your function will be very simple, but you will need to use `lambda`!

Now that we can create one row, we just need to create one row, many times.

Exercise

Write a function (times-table len) that returns the n×n times table.
Use times-row as a helper function.

```
(timestable 5) =>
  (list (list  1  2  3  4  5)
        (list  2  4  6  8 10)
        (list  3  6  9 12 15)
        (list  4  8 12 16 20)
        (list  5 10 15 20 25))
```

```
;; (times-table n) return the times table up to n*n.
;; times-table: Nat -> (listof (listof Nat))
;; Example:
(check-expect (times-table 3)
              (list (list 1 2 3) (list 2 4 6) (list 3 6 9)))
(check-expect (times-table 5) timestable5)
```

Higher order functions in many languages

map, **lambda**, etc. were introduced around 1958 in Lisp (of which Racket is a dialect), but are so useful that they have been added to many languages. Here are just a few examples:

language	code
Lisp, including Racket	`(map (lambda (x) (+ x 1)) (list 2 3 5 7 11))`
Python and Sage	`map(lambda x: x + 1, [2, 3, 5, 7, 11])`
Maple	`map(x -> x + 1, [2, 3, 5, 7, 11]);`
Haskell	`map (\x -> x + 1) [2, 3, 5, 7, 11]`
JavaScript	`[2, 3, 5, 7, 11].map(function (x) { return x + 1; })`
Matlab and GNU Octave	`arrayfun(@(x) (x + 1), [2, 3, 5, 7, 11])`
Perl	`map { $_ + 1 } (2, 3, 5, 7, 11);`

C++	`list<int> src = {2, 3, 5, 7, 11}, dest;`
	`transform(src.begin(), src.end(), dest.begin(),`
	`          [](int i) { return i + 1; });`

As you learn new languages, take these powerful tools with you!

When to use `list` and when to use `cons` ?

- If you are **creating** a new list of constant length, you may use **`list`**. For example,

```
(define oldlist (list 3 5 7))
oldlist => (list 3 5 7)
```

- If you are **expanding** an existing list, you must construct a larger list using **`cons`**.

```
(define newlist (cons 2 oldlist))
newlist => (list 2 3 5 7)
```

When to use `list` and when to use `cons` ?

What's the difference?

- `list` takes **any number** of arguments, and creates a list of exactly that length.
- `cons` always takes **exactly two** arguments: an `Any`, and another list, which may be the empty list, `'()`.

If you use `list` where you should use `cons`, you can get a list of length 2, that contains another list of length 2, that contains another list of length 2, that contains....

```
(foldr cons '() (list 2 3 5)) => (list 2 3 5)
(foldr list '() (list 2 3 5)) => (list 2 (list 3 (list 5 '())))
```
← This is bad!

> Except for creating examples, data, and other lists of known length, you should almost always use `cons` instead of `list`.

Exercise

Write a function `count-at` that consumes a `Str` and counts the number of times `#\a` or `#\t` appear in it.

```
count-at("A cat sat on a mat") => 7
```

This can be completed using **`foldr`** or **`filter`**. Try writing it both ways.

Module Summary

- Use `filter` to select only certain values from lists.
- Combine `filter` with `map`, `range`, and `foldr`.
- Use `cons` to construct lists. With `cons` and `foldr`, be able to manipulate lists without using `map` or `filter`.
- Be able to use `lambda`
  - To write short, single-use functions
  - To fill in extra parameters of helper functions

Further Reading: *How to Design Programs*, Section 8