Module 5: Deconstructing and constructing lists

If you have not already, make sure you

- Read the Wikipedia entry on *Filter (higher order function)*.
- Read *How to Design Programs*, Section 9.
Problem: Calculate the sum of all multiples of 2, 3, or 5, between 0 and 1000.

Maybe try something like:

```scheme
(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?

I can check a single number easily enough:

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

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

`; (multiple-235? n) return #true if n is divisible by 2, 3, or 5.
`; multiple-235?: Nat -> Bool`
Another higher order function: \texttt{filter}

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

\texttt{(filter F (list x\texttt{0} x\texttt{1} x\texttt{2} ... x\texttt{n}))} \Rightarrow \texttt{(list x\texttt{0} x\texttt{3} ... )}

For all values \texttt{x\texttt{k}} for which \texttt{(F x\texttt{k})} \texttt{\Rightarrow} \texttt{#true}.

\texttt{;; keep-multiples-235: (listof Nat) -> (listof Nat)}

\texttt{;; Example:}

\texttt{(check-expect (keep-multiples-235 (range 0 10 1)) (list 0 2 3 4 5 6 8 9))}

\texttt{(check-expect (keep-multiples-235 (list 2 4 7 7 50 4)) (list 2 4 50 4))}

\texttt{(define (keep-multiples-235 L) (filter multiple-235? L))}

\texttt{(define (keep-even L) (filter even? L))}

\texttt{(check-expect (keep-even (list 1 2 3 4 5 6)) (list 2 4 6))}
Here is an example of a function using `filter`: →

```
(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 12 7 30 3 19 42)) => (list 12 30 19)
```

**Exercise**

Use `filter` to write a function that keeps all items which are a `(list a b c)` containing a Pythagorean triple $a < b < c : a^2 + b^2 = c^2$

```
(check-expect (pythagoreans (list (list 1 2 3) (list 3 4 5) (list 5 12 13) (list 4 5 6)))
              (list (list 3 4 5) (list 5 12 13)))
```

**Hint**

You can use `first`, `second`, and `third` to get individual items from a list.
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.

\[
\text{(times-square (list 1 25 5 4 1 7))} \Rightarrow (* 1 25 4 1) \Rightarrow 100
\]

Hint
Use `integer?` to check if a value is an integer.
Two functions which operate on lists, and which we will use more later, are \texttt{first} and \texttt{rest}:

\begin{verbatim}
(define L (list 2 3 5 7 11))
(first L)   (rest L)
\end{verbatim}

\begin{verbatim}
\downarrow   \downarrow
2   (list 3 5 7 11)
\end{verbatim}

\texttt{first} consumes a \texttt{(listof Any)}, and returns the first value in that list. 
\texttt{rest} consumes a \texttt{(listof Any)}, and returns the list with all the values \texttt{except} the first.

We want to go the other way:

\begin{verbatim}
We may use \texttt{cons} to construct lists. It consumes two values: \texttt{Any}, and a \texttt{(listof Any)}. It returns a list one longer, with the new value added at the \texttt{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)
\end{verbatim}

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

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.

(remove-second (list 2 4 6 0 1)) => (list 2 6 0 1)
Using \texttt{foldr} to construct lists

Recall what \texttt{foldr} does:
\[
(\texttt{foldr} \ F \ \texttt{base} \ (\texttt{list} \ x_0 \ x_1 \ \ldots \ x_n)) \Rightarrow (F \ x_0 \ (F \ x_1 \ (F \ \ldots \ (F \ x_n \ \texttt{base}))))
\]

We can use \texttt{foldr} to copy a list.
\[
(\texttt{define} \ (\texttt{list-fun} \ a \ b) \ (\texttt{cons} \ a \ b))
\]

Recall \(a\) is a value from the list, and \(b\) is the accumulated value.

\[
(\texttt{foldr} \ \texttt{list-fun} \ () \ (\texttt{list} \ 2 \ 3 \ 5))
\]
\[
(\texttt{list-fun} \ 2 \ (\texttt{list-fun} \ 3 \ (\texttt{list-fun} \ 5 \ ()())())
\Rightarrow (\texttt{cons} \ 2 \ (\texttt{cons} \ 3 \ (\texttt{cons} \ 5 \ ()())())
\Rightarrow (\texttt{list} \ 2 \ 3 \ 5)
\]
Faking map

We can modify lists using `cons` and `foldr`, as if we were using `map`.

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

```scheme
;; (add-2 x) add 2 to x.
(define (add-2 x)
  (+ x 2))

;; (add-2-each L) add 2 to each of L.
(define (add-2-each L)
  (map add-2 L))
```

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

```scheme
;; (add-2-first a b) construct a list
;; using 2 more than a, then b.
(define (add-2-first a b)
  (cons (+ 2 a) b))

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

Write a function that consumes a `(listof Num)` and returns the list with each number doubled.

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

```scheme
(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

\[(\text{foldr } F \text{ base } (\text{list } x_0 \ x_1 \ldots \ xn)) \Rightarrow (F \ x_0 \ (F \ x_1 \ (F \ldots \ (F \ xn \ \text{base}))))\]

Exercise

Using \texttt{foldr}, write a function \((\text{keep-evens } L)\) that returns the list containing all the even values in \(L\).

That is, it acts like \((\text{filter even? } L)\).

\((\text{keep-evens } (\text{list } 1\ 2\ 3\ 4\ 5\ 6)) \Rightarrow (\text{list } 2\ 4\ 6)\)

Hint

With \texttt{foldr} you have the “partial answer” from the previous call, which here must be a \((\text{listof Num})\).

- Sometimes, you want to add the new value at the front of the old answer.
- Sometimes you want to ignore the new value, and just pass on the old answer.
Overview of Higher Order Functions

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

\[
\text{map } F \left( \text{list } x_0 \ x_1 \ \ldots \ x_n \right) \Rightarrow \left( \text{list } (F \ x_0) \ (F \ x_1) \ \ldots \ (F \ x_n) \right)
\]

\[
\text{map } \text{sqr} \left( \text{list } 2 \ 3 \ 5 \right) \Rightarrow \left( \text{list } 4 \ 9 \ 25 \right)
\]

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.

\[
\text{filter } G \left( \text{list } x_0 \ x_1 \ \ldots \ x_n \right) \Rightarrow \left( \text{list } x_0 \ x_2 \right), \text{if } x_0 \text{ and } x_2 \text{ are the only values in the list for which } G \text{ returns } #true.
\]

\[
\text{filter } \text{even?} \left( \text{list } 2 \ 5 \ 8 \ 7 \ 4 \ 3 \ 2 \right) \Rightarrow \left( \text{list } 2 \ 8 \ 4 \ 2 \right)
\]

foldr  Combine items in a list, and return a single value.

This could be of any type, even a list.

\[
\text{foldr } H \ \text{base} \left( \text{list } x_0 \ x_1 \ \ldots \ x_n \right) \Rightarrow \left( H \ x_0 \ (H \ x_1 \ (H \ \ldots \ (H \ x_n \ \text{base}))\right))
\]

\[
\text{foldr } * \ 7 \left( \text{list } 2 \ 10 \ 3 \right) \Rightarrow 420
\]
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**.

<table>
<thead>
<tr>
<th>desired output</th>
<th>likely solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>a list the same size as the input</td>
<td>try <strong>map</strong></td>
</tr>
<tr>
<td>a list containing some of the items from the input</td>
<td>try <strong>filter</strong></td>
</tr>
<tr>
<td>a single value</td>
<td>try <strong>foldr</strong></td>
</tr>
<tr>
<td>a list, but not something you can do with <strong>map</strong> and <strong>filter</strong></td>
<td>try <strong>foldr</strong>, using <strong>cons</strong></td>
</tr>
</tbody>
</table>

You may need to use some combination of these functions.
Data-driven design: some hints on how to use \texttt{foldr}

Recall what \texttt{foldr} does: 
\[(\texttt{foldr} \ F \ \texttt{base} \ (\texttt{list} \ x0 \ x1 \ \ldots \ xn)) \Rightarrow (F \ x0 \ (F \ x1 \ (F \ \ldots \ (F \ xn \ \texttt{base}))))\]

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

1. It says \((F \ x0 \ \ldots)\), \((F \ x1 \ \ldots)\), etc.
   So the first parameter has to be the same as the type of the values in the list.
2. It says \((F \ \ldots \ (F \ \ldots))\).
   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 \((F \ \ldots \ \texttt{base})\), so the \texttt{base} is also of this type.

Exercise

Given that \texttt{use-foldr} consumes a \((\texttt{listof Nat})\): 
\[(\texttt{define} \ (\texttt{use-foldr} \ L) \ (\texttt{foldr} \ \texttt{myfun} \ "\texttt{some-str}" \ L))\]

1. What is the contract for \texttt{myfun}?
2. What is the contract for \texttt{use-foldr}?

Exercise

Write a function \texttt{myfun} that allows \texttt{use-foldr} to do something.
Data-driven design: some hints on how to use foldr

Given that use-foldr consumes a \((\text{listof Nat})\):

\[
\text{(define (use-foldr L) (foldr myfun "some-str" L))}
\]

Since foldr is being used on a \((\text{listof Nat})\), the first parameter of myfun must be \text{Nat}:

\[
;; \text{myfun: Nat UnknownTypeX -> UnknownTypeX}
\]

Since the base is a \text{Str}, that must also be the type returned by myfun. So the whole contract is:

\[
;; \text{myfun: Nat Str -> Str}
\]

I'm going to see what happens with this function:

\[
\text{(define (myfun n s) (string-append (number->string n) s))}
\]

\[
\text{(foldr myfun base L)}
\]

What can we say about base and L?
A new command in a new language level

At this point we introduce a new command, \texttt{lambda}, which is not a part of the language we have used so far.

In Racket, select \texttt{Language $\rightarrow$ Choose language $\rightarrow$ 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**.

;; (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))

**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 that consumes a (listof Num) and returns a list containing the cube of each Num. \((x^3)\)
Practice with lambda

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** just once and **foldr** just once, and no [named] helper 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
Handling extra parameters with lambda

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

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

```scheme
;; (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)
```
Handling extra parameters with lambda

Earlier we had the following functions:

```scheme
(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:

```scheme
;; (keep-multiples n L) return all values in L which are divisible by n.
;; 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.
Handling extra parameters with `lambda`

Solution: use `lambda`.

```scheme
;; (keep-multiples n L) return all values in L which are divisible by n.
;; 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 n L)
  (filter (lambda (v) (divisible? v n)) 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 than `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)`
Write a function 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)))
Faking map with foldr using lambda

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

**Exercise**

Use `foldr` to write a function `(add-n-each n L)` that adds `n` to each value in `L`.

```
(add-n-each 7 (list 2 4 8)) => (list 9 11 15)
```

**Exercise**

Use `foldr` to write a function that behaves like `map`.

```
(my-map add1 (list 6 8 48)) => (list 7 9 49)
```
Using \textbf{foldr}, write a function \texttt{(keep-multiples n L)} that returns the list containing all the values in \texttt{L} that are multiples of \texttt{n}.

That is, it acts like \texttt{(filter (lambda (x) (= 0 (remainder x n))) L)}.

\texttt{(keep-multiples 3 \texttt{(list 1 2 3 4 5 6 7)) \to (list 3 6)}}

Use \textbf{foldr} to write a function that behaves like \texttt{filter}.

\texttt{(my-filter odd? \texttt{(list 4 5 9 6)) \to (list 5 9)}}
Lists of lists

There are no restrictions of the types of items that a list can contain. They can even contain other lists (and they more lists, and so on, as deep as we like).

This gives us a way to store data in a *table*. For example, the times table up to $5 \times 5$:

```
(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))
```
Using \texttt{map} with \texttt{range} we can only create a single list. How to create a list that contains lists?

Idea: write a function that uses \texttt{map} to create one row of the table. Then use this function inside another call to \texttt{map}.

**Exercise**

Write a function \((\text{times-row } n \text{ len})\) that returns the \(n\)th row of the times table. This should be a list of length \(\text{len}\). Write your function in the form \((\text{map } \ldots (\text{range } 1 (+ \text{len} 1) 1))\).

\((\text{times-row } 3 \text{ 4}) \Rightarrow (\text{list } 3 \ 6 \ 9 \ 12)\)

\((\text{times-row } 5 \text{ 3}) \Rightarrow (\text{list } 5 \ 10 \ 15)\)

**Hint**

Your function will be very simple, but you will need to use \texttt{lambda}!
Write a function `(times-table len)` that returns the $n \times n$ 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))
```
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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lisp, including Racket</td>
<td><code>(map (lambda (x) (+ x 1)) (list 2 3 5 7 11))</code></td>
</tr>
<tr>
<td>Python and Sage</td>
<td><code>map(lambda x: x + 1, [2, 3, 5, 7, 11])</code></td>
</tr>
<tr>
<td>Maple</td>
<td><code>map(x -&gt; x + 1, [2, 3, 5, 7, 11]);</code></td>
</tr>
<tr>
<td>Haskell</td>
<td><code>map (\x -&gt; x + 1) [2, 3, 5, 7, 11]</code></td>
</tr>
<tr>
<td>JavaScript</td>
<td><code>[2, 3, 5, 7, 11].map(function (x) { return x + 1; })</code></td>
</tr>
<tr>
<td>Matlab and GNU Octave</td>
<td><code>arrayfun(@(x) (x + 1), [2, 3, 5, 7, 11])</code></td>
</tr>
<tr>
<td>Perl</td>
<td><code>map { $_ + 1 } (2, 3, 5, 7, 11);</code></td>
</tr>
<tr>
<td>C++</td>
<td><code>list&lt;int&gt; src = {2, 3, 5, 7, 11}, dest;</code></td>
</tr>
<tr>
<td></td>
<td><code>transform(src.begin(), src.end(), dest.begin(), [](int i) { return i + 1; });</code></td>
</tr>
</tbody>
</table>
Str and higher order functions

A \texttt{(listof Str)} will work very well with higher order functions:

\begin{verbatim}
(foldr string-append "?" (list "hello" "how" "R" "U")) => "hellohowRU?"
\end{verbatim}

\begin{verbatim}
(filter (lambda (s) (odd? (string-length s))) (list "first" "second" "third" "fourth")) => (list "first" "third")
\end{verbatim}

\textbf{Exercise}

Write a function \texttt{acronymize} that consumes a \texttt{(listof Str)}, where each \texttt{Str} is of length at least 1, and returns a \texttt{Str} containing the first letter of each item in the list.

\texttt{(acronymize (list "Portable" "Network" "Graphics")) => "PNG"}
\texttt{(acronymize (list "GNU's" "Not" "UNIX")) => "GNU"}
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.
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 \texttt{cons} '() \texttt{(list 2 3 5)}) => \texttt{(list 2 3 5)}
(\foldr \texttt{list} '() \texttt{(list 2 3 5)}) => \texttt{(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**.
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

Before we begin the next module, please

- Read *How to Design Programs* Sections 11, 10, 12.