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?
I can check a single number easily enough.

The function `multiple-235?` produces `true` if a `Nat` is one of the numbers I need to add up:

```scheme
;; (divisible? n d) Determine if n is divisible by d.
;; divisible?: Nat Nat -> Bool
(define (divisible? n d) (= 0 (remainder n d)))
```

```scheme
;; (multiple-235? n) produce true if n is divisible by 2, 3, or 5.
;; multiple-235?: Nat -> Bool
(define (multiple-235? n)
  (or (divisible? n 2) (divisible? n 3) (divisible? n 5)))
```

```scheme
;; (keep-multiples-235 lon) keep all values in lon divisible by 2, 3, or 5.
;; Example:
(check-expect (keep-multiples-235 (range 0 10 1)) (list 0 2 3 4 5 6 8 9))
```

Somehow I need to keep only these numbers, and add them up.
Another higher order function: \textit{filter}

The built in function \textit{filter} does exactly what we need.

\textbf{(filter \ f \ lst)} consumes a predicate and a (listof \ Any). \textit{f} must be a one-parameter function that consumes the type(s) of value in \textit{lst}, and produces a \textit{Bool}.

\textbf{(filter \ f \ lst)} will produce a list containing all the items \textit{x} in \textit{loa} for which (\textit{f x}) produces \textit{true}.

\textbf{(filter \ f \ (list \ x0 \ x1 \ x2 \ \ldots \ \ xn))} \Rightarrow \textbf{(list \ x0 \ x3 \ \ldots \ )}

For all values \textit{x_k} for which (\textit{f x_k}) \Rightarrow \textit{true}. 
Another higher order function: **filter**

Here is a simple example using the built in predicate `even?`:

```
(define (keep-even loi) (filter even? loi))
(keep-even (list 0 1 2 3 4)) ⇒ (list 0 2 4)
```

Since `(even? 2) ⇒ true, (even? 4) ⇒ true, and (even? 6) ⇒ true`, but the rest produce `false`. 
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.

```scheme
;; (keep-short los) Keep all the values in los 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"))
```
Write a function \texttt{count-at} that consumes a \texttt{Str} and counts the number of times \texttt{\textbackslash a} or \texttt{\textbackslash t} appear in it.

\begin{verbatim}
(count-at "A cat sat on a mat") ⇒ 7
\end{verbatim}
Problem Solving with map, foldr, filter, and range

In combination, these functions are very powerful.

Exercise

Write a function **times-square** that consumes a (listof Nat) and produces the product of all the perfect squares (1, 4, 9, 16, 25, ... ) 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
Multi-argument \texttt{filter}?

We can use \texttt{map}, \texttt{foldr}, and \texttt{foldl} with multiple lists. Can we do the same with \texttt{filter}? Not directly. \texttt{filter} consumes a predicate and a \texttt{single (listof Any)}.

What would it do, anyway? It can't produce more than 1 list.

But maybe we can do something similar, by combining \texttt{map} and \texttt{filter}.

Here's the idea:

1. Combine the values, one from each list, into a single value.
2. Find the values we like using \texttt{filter}.
3. As needed, convert back to whatever we need.
Example: given a list of first names, and a list of last names, I want to keep the \textit{first name} of everyone whose \textit{last name} comes after "I".

\begin{verbatim}
(define gnames (list "Joseph" "Burt" "Douglas" "James" "David"))
(define snames (list "Hagey" "Matthews" "Wright" "Downey" "Johnston"))
;; (firstname-of-late-lastnames fnames lnames) produce a list containing the
;; first names from fname of people whose lasts from lnames comes after "I".
;; Examples:
(check-expect (firstname-of-late-lastnames gnames snames)
  (list "Burt" "Douglas" "David"))

;; firstname-of-late-lastnames: (listof Str) (listof Str) -> (listof Str)
;; Requires: fnames and lnames have the same length.
\end{verbatim}
Think about it, step by step. Run each step to check it.

1 Combine the values:

\[
\text{(map list gnames snames)}
\implies \text{(list (list "Joseph" "Hagey") (list "Burt" "Matthews") (list "Douglas" "Wright") (list "James" "Downey") (list "David" "Johnston"))}
\]

2 Now each item in the single list is a \text{(list Str Str)}. We need a predicate that consumes one of these, and says if the last name comes after "I":

\[
\text{(define (lname-after-I los)}
\text{(string>? (second los) "I"))}
\]

\[
\text{(filter lname-after-I (map list gnames snames))}
\implies \text{(list (list "Burt" "Matthews") (list "Douglas" "Wright") (list "David" "Johnston"))}
\]

3 Now grab the first name, and we’re done:

\[
\text{(map first (filter lname-after-I (map list gnames snames))})
\implies \text{(list "Burt" "Douglas" "David")}
\]
Exercise

Write a function `keep-bigger`. It consumes two `(listof Num)`. For each pair, it produces the second value only if it exceeds the first. For example:

```scheme
(check-expect (keep-bigger (list 1 6 8 0 5 7) (list 4 7 3 2 0 9)) (list 4 7 2 9))
```

Exercise

Write a function `even-location`. It consumes a `(listof Any)`, and produces a list containing only those values at an even-numbered location in the list. (In a list, we say that the first value is at location 0, the second is at location 1, and so on.)

```scheme
(check-expect (keep-even-index (list "I" "do" "not" "like" "green" "eggs" "and" "ham")) (list "I" "not" "green" "and"))
```
Two functions which operate on lists, and which we will use more later, are `first` and `rest`:

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

`first` consumes a `(listof Any)`, and produces the first value in that list.

`rest` consumes a `(listof Any)`, and produces the list with all the values except the first.
We want to go the other way:

We may use \texttt{cons} to construct lists:

- It consumes two values: an \texttt{Any}, and a \texttt{(listof Any)}.
- It produces a list one longer, with the new value added at the \textbf{left} of the list.

\[
\text{(cons 4 (list 1 2 3)) } \Rightarrow \text{(list 4 1 2 3)}
\]
\[
\text{(cons 1 (cons 2 (cons 3 empty))) } \Rightarrow \text{(list 1 2 3)}
\]

(\text{It's a little trickier to add to the right of a list, or to get the last item.})
Exercise
Construct \( (\text{list} \ 6 \ 7 \ 42) \) using only \textit{cons} and the empty list, \textit{empty}.

Exercise
Write a function \texttt{remove-second} that consumes a list of length at least two, and produces the same list with the second item removed.
\begin{verbatim}
(check-expect (remove-second (list 2 4 6 0 1)) (list 2 6 0 1))
\end{verbatim}
Using \texttt{foldr} to construct lists

Recall what \texttt{foldr} does:

\[
(\texttt{foldr } f \texttt{ base } (\texttt{list } x_0 \texttt{ x}_1 \ldots \texttt{x}_n)) \Rightarrow (f \ x_0 \ (f \ x_1 \ (f \ldots \ (f \ x_n \texttt{ base}))))
\]

We can use \texttt{foldr} to copy a list:

\[
(\texttt{foldr } \texttt{ cons } \texttt{ empty } (\texttt{list } 2 \texttt{ 3 } 5))
\Rightarrow (\texttt{cons } 2 \ (\texttt{cons } 3 \ (\texttt{cons } 5 \texttt{ empty})))
\Rightarrow (\texttt{list } 2 \texttt{ 3 } 5)
\]

How much more can we do with this?
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:

```scheme
;; (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 lon.
;; add-2-each-m: (listof Num) -> (listof Num)
(define (add-2-each-m lon)
  (map add-2 lon))
```

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

```scheme
;; (add-2-first newitem oldlist)
;; construct a list using 2 more
;; than newitem, then oldlist.
;; add-2-first: Num (listof Num) -> (listof Num)
(define (add-2-first newitem oldlist)
  (cons (+ 2 newitem) oldlist))

;; (add-2-each-f lon) +2 to each of lon.
(define (add-2-each-f lon)
  (foldr add-2-first empty lon))
```

Write more tests to verify that `add-2-each-f` works.
The function `double-each` works. Rewrite it using `foldr`, without using `map`.

```scheme
(define (double n) (* n 2))
(define (double-each lon) (map double lon))
```

```scheme
(foldr f base (list x0 x1 ... xn)) ⇒ (f x0 (f x1 (f ... (f xn base))))
```
We can create new lists using cons and foldr, as if we were using filter.

Recall that using filter, I can keep items bigger than 5:

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

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

We can do the same using only foldr.

Exercise
Copy the code from the commentary, then write more tests to verify that keep-big and keep-big-f both work.
Faking filter using foldr

\[
(foldr \ f \ base \ (list \ x0 \ x1 \ \ldots \ xn)) \Rightarrow (f \ x0 \ (f \ x1 \ (f \ \ldots \ (f \ xn \ base))))
\]

Exercise

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

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

\[
(\text{define} \ (\text{keep-evens} \ loi) \n\quad (\text{filter} \ \text{even?} \ loi))
\]

\[
(\text{check-expect} \ (\text{keep-evens} \ (\text{list} \ 1 \ 2 \ 3 \ 4 \ 5 \ 6)) \ (\text{list} \ 2 \ 4 \ 6))
\]

Hint

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

- Sometimes, you want to \text{cons} the new value to the old answer.
- Sometimes you want to ignore the new value, and just produce the old answer.
Overview of Higher Order Functions

map Transforms each item in a list, and produces 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 produces a list of the same items for which the predicate produces true. This list will usually be smaller.

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

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

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

This could be of any type, even a list.

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

\[ \text{foldr } * \ 7 \left( \text{list } 2 \ 10 \ 3 \right) \Rightarrow \underbrace{420} \]
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>consider <code>map</code></td>
</tr>
<tr>
<td>a list containing some of the items from the input</td>
<td>consider <code>filter</code></td>
</tr>
<tr>
<td>a single value</td>
<td>consider <code>foldr</code></td>
</tr>
<tr>
<td>a list, but not something you can do with <code>map</code> and <code>filter</code></td>
<td>consider <code>foldr</code>, using <code>cons</code></td>
</tr>
</tbody>
</table>

You may prefer to use some combination of these functions.
Recall what `foldr` does:

\[
(foldr \ f \ base \ (\text{list} \ x_0 \ x_1 \ ... \ x_n)) \Rightarrow (f \ x_0 \ (f \ x_1 \ (f \ ... \ (f \ x_n \ base))))
\]

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

1. It says \( (f \ x_0 \ ...) \), \( (f \ x_1 \ ...) \), 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 \( f \) produces will be used as the second parameter of \( f \).
   So the produced value and the second parameter must be of the same type.

3. It says \( (f \ ... \ base) \), so the \( base \) is also of this type.
Data-driven design: some hints on how to use \texttt{foldr}

That is, to write \((\texttt{foldr} \ f \ \texttt{base} \ \texttt{lox})\), where \texttt{lox} is a \((\text{listof } X)\), the contract for \(f\) must be of the form:

\[
f: X \ Y \rightarrow Y
\]

...and \texttt{base} must be of type \(Y\).

\begin{itemize}
  \item Given that \texttt{use-foldr} consumes a \((\text{listof } \text{Nat})\):
    \[
    (\textbf{define} \ (\texttt{use-foldr} \ \texttt{lon}) \ (\texttt{foldr} \ \texttt{myfun} \ "\text{some-str}" \ \texttt{lon}))
    \]
  \item Exercise: carefully consider:
    \begin{enumerate}
      \item What is the contract for \texttt{myfun} ?
      \item What is the contract for \texttt{use-foldr} ?
    \end{enumerate}
  \item Exercise: Write a function \texttt{myfun} that allows \texttt{use-foldr} to do something.
\end{itemize}
Consider this function:

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

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

- `n` must be a `Num` (since we pass it to `number->string`), so `lst` must be a `(listof Num)`.
- `s` must be a `Str` (since we pass it to `string-append`), so `base` must be a `Str`.
- It is good that `myfun` produces 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:

```scheme
;; (list-max lon) produce the largest value in lon.
(define (list-max lon)
  (foldr f base lon))
```

Exercise

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

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

```lisp
(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 produce if largest-so-far is 27, and new-item is 6?
- What should we produce 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.

```
(define (list-max lon)
  (foldr f 0 lon))
```

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.
Another thing we can do fairly easily with \texttt{foldr} is making one value change on or more of the values that came \textit{after} it in the list. (With \texttt{foldl} we can do similarly with the values that came \textit{before} it in the list.)

Idea: use \texttt{foldr} with some helper function \((f \text{ item ans})\). Given \texttt{pred?} is a predicate that produces \texttt{true} if its argument is the item we want to change after. Inside \(f\), write something like the following:

\begin{verbatim}
(define (f item ans)
  (cond [(pred? item) ; When item is the kind we want...
                (cons (g (first ans)) ; change the value *after* it.
                      (rest ans))]
        [else (cons item ans)])) ; Otherwise, add new value at front.
\end{verbatim}
**Mucking with adjacent values**

**Exercise**

Change ponder so `muck-after-str` changes every value that immediately follows the word "SQUARE" to be the square of that number, and it removes all `Str`.

(check-expect (muck-after-str (list 5 "yo" 7 "SQUARE" 4 3)) (list 5 7 16 3))

**Exercise**

Change ponder so `muck-after-str` also removes every value that immediately follows the word "POP".

(check-expect (muck-after-str (list 5 8 "POP" 4 3)) (list 5 8 3))

Read the documentation on the `empty?` predicate. Use this predicate to allow the function to ignore "POP" when it appears at the very end of the list.

(check-expect (muck-after-str (list 5 8 4 3 "POP")) (list 5 8 4 3))

**Exercise**

Also make the word "ADD" add up the two values that come after it.

(check-expect (muck-after-str (list 5 8 4 3 "POP")) (list 5 8 4 3))

(check-expect (muck-after-str (list 5 8 "ADD" 7 3 5)) (list 5 8 10 5))
Write a function `sum-pieces` that consumes a `(listof (anyof Nat #\!))`, and produces a `(listof Nat)`. The `#\!` separate the items into groups. The function collects each “group” of Nat together into a sum.

For example:

- `(sum-pieces (list 2 3 #\! 7 4 #\! 10 15)) ⇒ (list 5 11 25); 3 groups`
- `(sum-pieces (list 2 3 1 1 1)) ⇒ (list 8); one group`
- `(sum-pieces (list 4 2 5 #\!)) ⇒ (list 11 0); one empty group`
- `(sum-pieces (list #\!)) ⇒ (list 0 0); two empty groups`
At this point we introduce a new command, \texttt{lambda}, which is not a part of the language we have used so far.
Some simple things are annoying

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

;; (double n) produce 2*n.
;; Examples:
(check-expect (double 3) 6)

;; double: Num -> Num
(define (double n) (* n 2))

;; (double-each lon) produce lon, with each value doubled.
;; Examples:
(check-expect (double-each (list 2 3 5)) (list 4 6 10))

;; double-each: (listof Num) -> (listof Num)
(define (double-each lon) (map double lon))

Half the work is the design recipe for a really simple function!
For short functions which are used just once, \texttt{lambda} lets us write \textbf{anonymous functions}.

An example:

\texttt{;; (double-each2 lon) produce lon, with each value doubled.}

\texttt{;; Examples:}

\texttt{(check-expect (double-each2 (list 2 3 5)) (list 4 6 10))}

\texttt{;; double-each2: (listof Num) -> (listof Num)}

\texttt{(define (double-each2 lon)}

\texttt{  (map (lambda (n) (* n 2)) lon))}

Remember: the first parameter to \texttt{map} is a function.

Here \texttt{(lambda (n) (* n 2))} takes the place of the function.

That \texttt{lambda} expression \textbf{is} a function.
**Tiny Functions with **$\texttt{lambda}$**

$\texttt{lambda}$ is a special form that produces a function.

$\langle \texttt{lambda } (x) \ ( + \ x \ 7) \rangle$ is a function with one parameter.

$\langle \texttt{map } \langle \texttt{lambda} \ (x) \ ( + \ x \ 7) \rangle \ \langle \texttt{list} \ 2 \ 3 \ 5 \rangle \rangle \Rightarrow \langle \texttt{list} \ 9 \ 10 \ 12 \rangle$

**Exercise**

Using $\texttt{lambda}$ and $\texttt{map}$, but no [named] helper functions, write a function $\texttt{cube-each}$ that consumes a $\langle \texttt{listof} \ \texttt{Num} \rangle$ and produces a list containing the cube of each $\texttt{Num}$: $\langle x^3 \rangle$

$\langle \texttt{check-expect} \ (\texttt{cube-each} \ \langle \texttt{list} \ 1 \ 2 \ 3 \rangle) \ \langle \texttt{list} \ 1 \ 8 \ 27 \rangle \rangle$
Using **lambda** and **filter** but no named helper functions, write a function that consumes a `(listof Str)` and produces a list containing all the strings that have a length of 4.

(\text{keep4 } (\text{list } "\text{There's}\" "\text{no}\" "\text{fate}\" "\text{but}\" "\text{what}\" "\text{we}\" "\text{make}\" "\text{for}\" "\text{ourselves}\")})

⇒ (\text{list } "\text{fate}\" "\text{what}\" "\text{make}\")

Using **lambda** but no named helper functions, write a function that consumes a `(listof Int)` and produces the sum of all the values divisible by 3.

(\text{sum-3s } (\text{list } 2 3 4 5 6)) ⇒ 9

Can you do it using **lambda** just once and **foldr** just once?
Handling extra parameters with \texttt{lambda}

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

\begin{verbatim}
(define (add-5 n) (+ n 5))
(define (add-5-each lon) (map add-5 lon))
\end{verbatim}

(\texttt{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. I want to write a function to add a given value to each item in a list. Like so:

\begin{verbatim}
(add-n-each (list 3.2 6 8) 6) ⇒ (list 9.2 12 14)
(add-n-each (list 3.2 6 8) 2) ⇒ (list 5.2 8 10)
\end{verbatim}

There’s a problem: if I add a parameter \texttt{n} to \texttt{add-5-each}, I don’t have a way for that value to be available to \texttt{add-5}.
Handling extra parameters with **lambda**

We can fix it using **lambda**!

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

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

**Exercise**

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

```scheme
(multiply-each (list 2 3 5) 4) ⇒ (list 8 12 20)
```

**Exercise**

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

```scheme
(add-total (list 2 3 5 10)) ⇒ (list 22 23 25 30)
```
A few details about \textit{lambda}

Using \textit{lambda} expression we can create a constant which stores a function.

\begin{verbatim}
(define double (lambda (x) (* 2 x)))
\end{verbatim}

\begin{verbatim}
(double 5) \Rightarrow 10
\end{verbatim}

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

You can use a \textit{lambda} expression anywhere you need a function:

\begin{verbatim}
((lambda (x y) (+ x y y)) 2 5) \Rightarrow 12
\end{verbatim}

Anything that can go in a function can go in a \textit{lambda}, even another \textit{lambda}:

\begin{verbatim}
((lambda (x y)
   ((lambda (z) (+ x z)) y)) 4 5)
\end{verbatim}
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 lon) (filter multiple-235? lon))
```

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

```scheme
;; (keep-multiples d lon) produce all values in lon 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 \texttt{lambda}.

\begin{verbatim}
;; (keep-multiples d lon) produce all values in lon 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 lon)
    (filter (lambda (v) (divisible? v d)) lon))
\end{verbatim}

The \texttt{n} and \texttt{lon} variables are \textbf{in scope} inside the \texttt{lambda} function. It can use them!
Write \((\text{discard-bad} \; \text{lon} \; \text{lo} \; \text{hi})\). It consumes a \((\text{listof Num})\) and two \text{Num}. It produces the list of all values in \text{lon} that are between \text{lo} and \text{hi}, inclusive.

\[(\text{discard-bad} \; \text{list} \; 12 \; 5 \; 20 \; 2 \; 10 \; 22) \; 10 \; 20) \Rightarrow \text{list} \; 12 \; 20 \; 10\]

Write \((\text{squash-bad} \; \text{lo} \; \text{hi} \; \text{lon})\). It consumes two \text{Num} and a \((\text{listof Num})\). Values in \text{lon} that are greater that \text{hi} become \text{hi}; less that \text{lo} become \text{lo}.

\[(\text{squash-bad} \; 10 \; 20 \; \text{list} \; 12 \; 5 \; 20 \; 2 \; 10 \; 22) \Rightarrow \text{list} \; 12 \; 10 \; 20 \; 10 \; 10 \; 20\]
Write a function `above-average` that consumes a `(listof Num)` and produces the list containing just the values which are greater than or equal to the average (mean) value in the list.
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`.
We want to be able to make a times table, something like the following:

```
(timestable 4) ⇒
(list (list 1 2 3 4)
     (list 2 4 6 8)
     (list 3 6 9 12)
     (list 4 8 12 16))
```

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 produces the \( n \)th row of the times table. This should be a list of length `len`. Write your function in the form

```
(map ... (range 1 (+ len 1) 1)).
```

(check-expect (times-row 3 4) (list 3 6 9 12))
(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 produces the $n \times n$ times table. Use `times-row` as a helper function.

```
(timestable 4) ⇒
  (list (list 1 2 3 4)
       (list 2 4 6 8)
       (list 3 6 9 12)
       (list 4 8 12 16))
```

`; (times-table n) produce 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)))`
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>Scheme, including Racket</td>
<td>(map (lambda (x) (+ x 1)) (list 2 3 5 7 11))</td>
</tr>
<tr>
<td>Python and Sage</td>
<td>map(lambda x: x + 1, [2, 3, 5, 7, 11])</td>
</tr>
<tr>
<td>Maple</td>
<td>map(x -&gt; x + 1, [2, 3, 5, 7, 11]);</td>
</tr>
<tr>
<td>Haskell</td>
<td>map (\x -&gt; x + 1) [2, 3, 5, 7, 11]</td>
</tr>
<tr>
<td>Maple</td>
<td>map(x -&gt; x + 1, [2, 3, 5, 7, 11]);</td>
</tr>
<tr>
<td>Maple</td>
<td>arrayfun(@(x) (x + 1), [2, 3, 5, 7, 11])</td>
</tr>
<tr>
<td>Perl</td>
<td>map { $_ + 1 } (2, 3, 5, 7, 11);</td>
</tr>
<tr>
<td>C++</td>
<td>list&lt;int&gt; src = {2, 3, 5, 7, 11}, dest;</td>
</tr>
<tr>
<td></td>
<td>transform(src.begin(), src.end(), dest.begin(),</td>
</tr>
<tr>
<td></td>
<td>[](int i) { return i + 1; });</td>
</tr>
</tbody>
</table>

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,
  
  ```lisp
  (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`.
  
  ```lisp
  (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, `empty`.

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 empty (list 2 3 5)) ⇒ (list 2 3 5)
(foldr list empty (list 2 3 5)) ⇒ (list 2 (list 3 (list 5 empty))) ← Bad!
```

Except for creating examples, data, and other lists of known length, you should almost always use `cons` instead of `list`.
Racket allows one to define and use **symbols** with meaning to us (not to Racket).

A symbol is defined using an apostrophe or ‘quote’: 

\[
'CS135
\]

`CS135` is a value just like 0 or 135, but it is more limited computationally.

Symbols allow a programmer to avoid using constants to represent names of courses, colours, planets, or types of music.
Symbols can be compared using the predicate `symbol=?`.

```
(define home 'Earth)

(symbol=? home 'Mars) ⇒ false
```

`symbol=?` is the only function we’ll use in CS135 that is applied only to symbols. Other functions can be applied to many different types, including symbols.

Unlike numbers, symbols are self-documenting – you don’t need to define constants for them.
First create a constant:

(define mysymbol 'blue)

Then see what each of these expressions evaluates to:

(symbol=? mysymbol 'blue) ⇒ true
(symbol=? mysymbol 'red) ⇒ false
(symbol=? mysymbol 42) ⇒ error
(symbol? mysymbol) ⇒ true
(symbol? '*@) ⇒ true
(symbol? "the artist formerly known as Prince") ⇒ false
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
In this module we added the following to our toolbox:
cons eighth empty? fifth filter fourth lambda second seventh sixth symbol=? symbol? third

These are the functions and special forms currently in our toolbox:
  check-expect check-within cond cons define eighth else empty? even? exp expt fifth
  filter first floor foldl foldr fourth integer? lambda length list list->string list?
  map max member? min not number->string number? odd? or quotient range remainder rest
  second seventh sixth sqr sqrt string->list string-append string-length string<=?