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.

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)))

;; (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)))

;; (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: `filter`

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

```scheme
(filter f lst) consumes a predicate and a (listof Any). f must be a one-parameter function that consumes the type(s) of value in lst, and produces a Bool.
(filter f lst) will produce a list containing all the items x in lon for which (f x) produces true.

(filter f (list x0 x1 x2 ... xn)) ⇒ (list x0 x3 ... )
```

For all values `xk` for which `(f xk)` ⇒ `true`. 
Another higher order function: **filter**

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

```scheme
(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`.

**filter practice**

**Exercise**

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

```scheme
(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"))
```

**filter practice**

**Exercise**

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

```scheme
(count-at "A cat sat on a mat") ⇒ 7
```
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)) \(\Rightarrow\) \((\ast 1 25 4 1)\) \(\Rightarrow\) 100

**Multi-argument filter?**

We can use `map`, `foldr`, and `foldl` with multiple lists. Can we do the same with `filter`?

Not directly. `filter` consumes a predicate and a 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 `map` and `filter`.

Here's the idea:

1. Combine the values, one from each list, into a single value.
2. Find the values we like using `filter`.
3. As needed, convert back to whatever we need.

**Multi-argument filter?**

Example: given a list of first names, and a list of last names, I want to keep the first name of everyone whose last name comes after "I".

```
(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) \(\rightarrow\) (listof Str)
;; Requires: fnames and lnames have the same length.
Think about it, step by step. Run each step to check it.

1 Combine the values:
\[
(map \ list \ gnames \ snames) \\
\Rightarrow (\text{list} \ \text{list} \ "Joseph" \ "Hagey") \ (\text{list} \ "Burt" \ "Matthews") \ (\text{list} \ "Douglas" \ "Wright") \ (\text{list} \ "James" \ "Downey") \ (\text{list} \ "David" \ "Johnston")
\]

2 Now each item in the single list is a (list Str Str). We need a predicate that consumes one of these, and says if the last name comes after "I":
\[
\begin{align*}
\text{define} & \ \ (\text{lname-after-I} \ los) \\
& \ \ (\text{string>?} \ (\text{second} \ los) \ "I") \\
(\text{filter} \ \text{lname-after-I} \ (map \ list \ gnames \ snames)) \\
\Rightarrow & \ \ (\text{list} \ \text{list} \ "Burt" \ "Matthews") \ (\text{list} \ "Douglas" \ "Wright") \ (\text{list} \ "David" \ "Johnston")
\end{align*}
\]

3 Now grab the first name, and we’re done:
\[
(\text{map} \ \text{first} \ (\text{filter} \ \text{lname-after-I} \ (\text{map} \ \text{list} \ gnames \ snames))) \\
\Rightarrow & \ \ (\text{list} \ "Burt" \ "Douglas" \ "David")
\]

---

Practice

**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:
\[
\begin{align*}
\text{check-expect} & \ \ (\text{keep-bigger} \ (\text{list} \ 1 \ 6 \ 8 \ 0 \ 5 \ 7) \\
& \ \ (\text{list} \ 4 \ 7 \ 3 \ 2 \ 0 \ 9)) \\
& \ \ (\text{list} \ 4 \ 7 \ 2 \ 9)
\end{align*}
\]

Write a function keep-even-index. 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.)
\[
\begin{align*}
\text{check-expect} & \ \ (\text{keep-even-index} \\
& \ \ (\text{list} \ "I" \ "do" \ "not" \ "like" \ "green" \ "eggs" \ "and" \ "ham")) \\
& \ \ (\text{list} \ "I" \ "not" \ "green" \ "and")
\end{align*}
\]

---

Constructing lists

Two functions which operate on lists, and which we will use more later, are first and rest:
\[
(\text{define} \ \text{lon} \ (\text{list} \ 2 \ 3 \ 5 \ 7 \ 11)) \\
(\text{first} \ \text{lon}) \ \ (\text{rest} \ \text{lon}) \\
\downarrow \hspace{1cm} \downarrow \\
2 \hspace{1cm} (\text{list} \ 3 \ 5 \ 7 \ 11)
\]

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 `cons` to construct lists:

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

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

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

---

### Constructing lists

**Exercise**

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

**Exercise**

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

\[
\text{\texttt{(check-expect (remove-second (list 2 4 6 0 1)) (list 2 6 0 1)}}
\]

---

### Using `foldr` to construct lists

Recall what `foldr` does:

\[
\text{\texttt{(foldr f base (list x0 x1 ... xn)) ⇒ (f x0 (f x1 (f ... (f xn base))))}}
\]

We can use `foldr` to copy a list:

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

How much more can we do with this?
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:

```scheme
(define (add-2 x)
 (+ x 2))
```

I can do the same thing with foldr instead:

```scheme
(define (add-2-first newitem oldlist)
 (cons (+ 2 newitem) oldlist))
```

Write more tests to verify that add-2-each-f works.

Faking map

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))))
```

Faking filter using foldr

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
(define (keep-big lon)
 (filter big? lon))
```

```scheme
(define (big? x)
 (> x 5))
```

We can do the same using only foldr.

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 ... xn)) ⇒ (f x0 (f x1 (f ... (f xn base)))

Using foldr, write a function (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:

(define (keep-evens loi)
  (filter even? loi))

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

Exercise

Using foldr, write a function (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:

(define (keep-evens loi)
  (filter even? loi))

(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 produce the old answer.

Overview of Higher Order Functions

map Transforms each item in a list, and produces 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 produces a list of the same items for which the predicate produces 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 produces true.

(filter even? (list 2 5 8 7 4 3 2)) ⇒ (list 2 8 4 2)

foldr Combine items in a list, and produce 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

Data-driven design

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 map</td>
</tr>
<tr>
<td>a list containing some of the items from the input</td>
<td>consider filter</td>
</tr>
<tr>
<td>a single value</td>
<td>consider foldr</td>
</tr>
<tr>
<td>a list, but not something you can do with map and filter</td>
<td>consider foldr, using cons</td>
</tr>
</tbody>
</table>

You may prefer to use some combination of these functions.
Data-driven design: contracts and foldr 22/53

Recall what `foldr` does:

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

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 \(f\) 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 foldr 23/53

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

\[
f: X \ Y \Rightarrow Y
\]

...and base must be of type Y.

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

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

carefully consider:

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

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

Data-driven design: some hints on how to use foldr 24/53

Consider this function:

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

What can we say about base and \(\lst\)?

- \(n\) must be a Num (since we pass it to \(\text{number->string}\)), so \(\lst\) must be a \((listof \text{Num})\).
- \(s\) must be a Str (since we pass it to \(\text{string-append}\)), so base number be a Str.
- It is good that myfun produces a value of the same type as s.
More with \texttt{foldr}

We haven't yet seen the whole power of \texttt{foldr}.

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

I can use \texttt{foldr} to get the largest value, something like this:
\begin{verbatim}
;; (list-max lon) produce the largest value in lon.
(define (list-max lon)
  (foldr f base lon))
\end{verbatim}

Exercise
What is the contract for \texttt{list-max}?
What is type of \texttt{base}?
What is the contract for \texttt{f}?

Since the final answer is a \texttt{Num},

- We have
  \begin{verbatim}
  ;; list-max: (listof Num) -> Num
  \end{verbatim}
- \texttt{base} must be a \texttt{Num}.
- We have
  \begin{verbatim}
  ;; f: Num Num -> Num
  \end{verbatim}

\texttt{base} needs to be some \texttt{Num}. We need to think about what \texttt{Num}, but for now, just use 0.

Let's use some helpful variable names in defining \texttt{f}. We have:
\begin{verbatim}
(f new-item largest-so-far) consumes two Num.
\end{verbatim}

- \texttt{new-item} is an item from the list.
- \texttt{largest-so-far} is the largest item we have found so far.

Two examples to consider:
- What should we produce if \texttt{largest-so-far} is 27, and \texttt{new-item} is 6?
- What should we produce if \texttt{largest-so-far} is 27, and \texttt{new-item} is 42?

Replace \texttt{base} with 0.

Exercise
Write \texttt{f} so \texttt{list-max} works, at least for some inputs.
\begin{verbatim}
(define (list-max lon)
  (foldr f base lon))
\end{verbatim}

Exercise
You may have a bug in your code. Try out the following test:
\begin{verbatim}
(check-expect (list-max (list -3 -17 -5)) -3)
\end{verbatim}
Then change \texttt{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 \((f \text{ item} \text{ ans})\). Given \texttt{pred?} is a predicate that produces \texttt{true} if its argument is the item we want to change after. Inside \(r\), 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}

Exercise

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

\begin{verbatim}
(check-expect (muck-after-str (list 5 "yo" 7 "SQUARE" 4 3)) (list 5 7 16 3))
\end{verbatim}

Exercise

Change \texttt{ponder} so \texttt{muck-after-str} also removes every value that immediately follows the word \texttt{"POP"}.

\begin{verbatim}
(check-expect (muck-after-str (list 5 8 "POP" 4 3)) (list 5 8 3))
\end{verbatim}

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

\begin{verbatim}
(check-expect (muck-after-str (list 5 8 4 3 "POP")) (list 5 8 4 3))
\end{verbatim}

Exercise

Also make the word \texttt{"ADD"} add up the two values that come after it.

\begin{verbatim}
(check-expect (muck-after-str (list 5 8 4 3 "ADD")) (list 5 8 4 3))
(check-expect (muck-after-str (list 5 8 "ADD" 7 3 5)) (list 5 8 10 5))
\end{verbatim}

Exercise

Write a function \texttt{sum-pieces} that consumes a \((\text{listof (anyof Nat \#\!)})\), and produces a \((\text{listof Nat})\).

The \#\! separate the items into groups. The function collects each "group" of \texttt{Nat} together into a sum.

For example:

\begin{verbatim}
(sum-pieces (list 2 3 #\! 7 4 #\! 10 15)) ⇒ (list 5 11 25) ; 3 groups
(sum-pieces (list 2 3 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
\end{verbatim}
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.

Some simple things are annoying

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

```scheme
;; (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!

Tiny Functions with `lambda`

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

An example:

```scheme
;; (double-each2 lon) produce lon, with each value doubled.
;; Examples:
(check-expect (double-each2 (list 2 3 5)) (list 4 6 10))

;; double-each2: (listof Num) -> (listof Num)
(define (double-each2 lon)
  (map (lambda (n) (* n 2)) lon))
```

Remember: the first parameter to `map` is a function.

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 produces a function.

\((\text{lambda} \,(x) \,(+ \,x \,7))\) is a function with one parameter.

\((\text{map} \,(\text{lambda} \,(x) \,(+ \,x \,7)) \,(\text{list} \,2 \,3 \,5)) \Rightarrow \,(\text{list} \,9 \,10 \,12)\)

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

(check-expect (cube-each (list 1 2 3)) (list 1 8 27))

**Practice with lambda**

Using lambda and filter but no named helper functions, write a function that consumes a \((\text{listof} \,\text{Str})\) and produces 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")

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

(sum-3s (list 2 3 4 5 6)) ⇒ 9

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

**Handling extra parameters with lambda**

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

(define (add-5 n) (+ n 5))

(define (add-5-each lon) (map add-5 lon))

(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:

(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)

There’s a problem: if I add a parameter \(n\) to add-5-each, I don’t have a way for that value to be available to add-5.
We can fix it using \texttt{lambda}!

\begin{verbatim}
;; (add-n-each lon n) add n to each item in lon.
;; add-n-each: (listof Num) Num -> (listof Num)
(define add-n-each (lambda (lon n)
                     (map (lambda (x) (+ x n))
                          lon)))
\end{verbatim}

This \texttt{lambda} expression, since it is inside \texttt{add-n-each}, can use the value of \texttt{n}. \texttt{n} is \textit{in scope}.

\begin{itemize}
  \item \textbf{Exercise}
    \begin{itemize}
      \item Write a function \texttt{(multiply-each lon n)}. It consumes a \texttt{(listof Num)} and a \texttt{Num}, and produces the list containing all the values in \texttt{lon}, each multiplied by \texttt{n}.
      \begin{verbatim}
      (multiply-each (list 2 3 5) 4) => (list 8 12 20)
      \end{verbatim}
    \end{itemize}
  \item \textbf{Exercise}
    \begin{itemize}
      \item Write a function \texttt{(add-total lon)} that consumes a \texttt{(listof Num)}, and adds the total of the values in \texttt{lon} to each value in \texttt{lon}.
      \begin{verbatim}
      (add-total (list 2 3 5 10)) => (list 22 23 25 30)
      \end{verbatim}
    \end{itemize}
\end{itemize}

\section*{A few details about \texttt{lambda}}

Using \texttt{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) => 10
\end{verbatim}

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

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

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

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

\begin{verbatim}
((lambda (x y) ((lambda (z) (+ x z)) y)) 4 5)
\end{verbatim}

\section*{Handling extra parameters with \texttt{lambda}}

Earlier we had the following functions:

\begin{verbatim}
(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))
\end{verbatim}

Suppose I wanted to keep multiples of a \texttt{Nat} which is a parameter:

\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))
\end{verbatim}

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

\textbf{Exercise}
Write \texttt{(discard-bad lon lo hi)}. It consumes a \texttt{(listof Num)} and two \texttt{Num}. It produces the list of all values in \texttt{lon} that are between \texttt{lo} and \texttt{hi}, inclusive.

\texttt{(discard-bad (list 12 5 20 2 10 22) 10 20)} ⇒ \texttt{(list 12 20 10)}

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

\texttt{(squash-bad 10 20 (list 12 5 20 2 10 22)) ⇒ (list 12 10 20 10 10 20)}

\textbf{More Practice}

\textbf{Exercise}
Write a function \texttt{above-average} that consumes a \texttt{(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 \(n \times n\).
;; 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><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; transform(src.begin(), src.end(), dest.begin(), [](int i) { return i + 1; });</code></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,
  ```scheme```
  ```(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.
  ```scheme```
  ```(define newlist (cons 2 oldlist))```
  ```newlist ⇒ (list 2 3 5 7)```

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....

```scheme```
```(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.
Symbolic data

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.

**Exercise**

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
```
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

Summary: built-in 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: