Module 5: Deconstructing and constructing lists

If you have not already, please

- Read the Wikipedia entry on Filter (higher order function).
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?` returns `#true` if a `Nat` is one of the numbers I need to add up:

\[
\text{(define (divisible? n d) (= 0 (remainder n d)))}
\]

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

\[
\text{(define (multiple-235? n)}
\text{\hspace{1cm} (or (divisible? n 2) (divisible? n 3) (divisible? n 5)))}
\]

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.

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

\[(\text{filter } F \ (\text{list } x_0 \ x_1 \ x_2 \ ... \ x_n)) \rightarrow (\text{list } x_0 \ x_3 \ ... \ )\]

For all values \(x_k\) for which \((F \ x_k) \Rightarrow \text{#true}\).
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))

Here is an example of a function using `filter`: 

\[
\text{(define (not-apple x) (not (equal? x "apple")))}
\]

\[
\text{(define (drop-apples L) (filter not-apple L))}
\]

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"))
Problem Solving with map, foldr, filter, and range

In combination, these functions are very powerful.

Write a function `times-square` that consumes a `(listof Nat)` and returns 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

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

(integer? (sqrt 5)) => #false
(integer? (sqrt 4)) => #true
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 \texttt{cons} to construct lists:

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

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

(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.

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

Recall what \texttt{foldr} does:

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

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

\[
(\texttt{foldr} \ \texttt{cons} \ () \ (\texttt{list} \ 2 \ 3 \ 5))
\Rightarrow (\texttt{cons} \ 2 \ (\texttt{cons} \ 3 \ (\texttt{cons} \ 5 \ ()())))
\Rightarrow (\texttt{list} \ 2 \ 3 \ 5)
\]
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:
```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 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:
```scheme
;; (add-2-first a b) construct 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))
```

Write more tests to verify that `add-2-each-m` and `add-2-each-f` both work.
The following function works. Rewrite it using \texttt{foldr}, without using \texttt{map}.

\texttt{(define (double n) (* n 2))}

\texttt{(define (double-each L) (map double L))}

\texttt{(foldr F base (list x0 x1 \ldots \ xn)) \Rightarrow (F x0 (F x1 (F \ldots (F xn base))))}
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

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

Exercise

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

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

\[
\text{define (keep-evens } L)
\text{(filter even? } L))
\]

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

Hint

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

- Sometimes, you want to 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**.

\[
(map \ F \ (list \ x_0 \ x_1 \ ... \ x_n)) \Rightarrow (list \ (F \ x_0) \ (F \ x_1) \ ... \ (F \ x_n))
\]

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

**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.

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

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

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

This could be of any type, even a list.

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

\[
(foldr \ * \ 7 \ (list \ 2 \ 10 \ 3)) \Rightarrow 420
\]
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>consider <strong>map</strong></td>
</tr>
<tr>
<td>a list containing some of the items from the input</td>
<td>consider <strong>filter</strong></td>
</tr>
<tr>
<td>a single value</td>
<td>consider <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>consider <strong>foldr</strong>, using <strong>cons</strong></td>
</tr>
</tbody>
</table>

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

Recall what \texttt{foldr} does:

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

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

1. It says \((F \ x_0 \ \ldots)\), \((F \ x_1 \ \ldots)\), 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 \((F \ \ldots \ \text{base})\), so the base is also of this type.
Data-driven design: some hints on how to use `foldr`

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

\[ F: X \ Y \rightarrow \ Y \]

...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:

```scheme
(define (myfun n s) (string-append (number->string n) s))
```

```scheme
(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`.
More with foldr

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 \texttt{Num},

- We have
  \[ ;; \text{list-max}: \text{(listof Num)} \rightarrow \text{Num} \]
- base must be a \texttt{Num}.
- We have
  \[ ;; F: \text{Num} \text{ Num} \rightarrow \text{Num} \]

base needs to be some \texttt{Num}. We need to think about what \texttt{Num}, but for now, just use \texttt{0}. 
Let’s use some helpful variable names in defining $F$. We have:

$F \text{ new-item largest-so-far}$ consumes two $\text{Num}$.

- 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

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

\begin{itemize}
  \item In Racket, select \textit{Language} $\rightarrow$ \textit{Choose language} $\rightarrow$ \textit{Intermediate student with lambda}.
\end{itemize}

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:

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

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

```scheme
;; (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))
```

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

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:

;;; (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`. $(x^3)$
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.

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

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

\[
\text{sum-evens (list 2 3 4 5)} \Rightarrow 6
\]

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:

;;; (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)
A few details about lambda

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:

\[
\text{(define (divisible? n d) (= 0 (remainder n d)))}
\]

\[
\text{(define (multiple-235? n) (or (divisible? n 2) (divisible? n 3) (divisible? n 5)))}
\]

\[
\text{(define (keep-multiples-235 L) (filter multiple-235? L))}
\]

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

\[
\text{;; (keep-multiples d L) return all values in L which are divisible by d.}
\]

\[
\text{;; keep-multiples: Nat (listof Nat) -> (listof Nat)}
\]

\[
\text{;; Examples:}
\]

\[
\text{(check-expect (keep-multiples 7 (list 2 3 5 28 7 3 14 77)) (list 28 7 14 77))}
\]

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

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 \texttt{n} and \texttt{L} variables are \texttt{in scope} inside the \texttt{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)))
```
Building tabular data

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 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\textsuperscript{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.

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))

;; (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:

<table>
<thead>
<tr>
<th>language</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lisp, 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>JavaScript</td>
<td>[2, 3, 5, 7, 11].map(function (x) { return x + 1; })</td>
</tr>
<tr>
<td>Matlab and GNU Octave</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; transform(src.begin(), src.end(), dest.begin(), [](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,
  ```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)
  ```
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....

```plaintext
(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**.
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