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

```scheme
(define (multiple-235? n)
  (or (divisible? n 2) (divisible? n 3) (divisible? n 5)))
```
Another higher order function: filter

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

(filter F (list x0 x1 x2 ... xn)) ⇒ (list x0 x3 ... )
For all values \( x_k \) for which \( (F x_k) \Rightarrow \#true \).

;; 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))

(define (keep-even L) (filter even? L))
(check-expect (keep-even (list 1 2 3 4 5 6)) (list 2 4 6))
Here is an example of a function using `filter`:

```scheme
(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.

```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 keeps all items which are a `(list a b c)` containing a Pythagorean triple \( a < b < c : a^2 + b^2 = c^2 \)

```scheme
(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.
Problem Solving with \textit{map}, \textit{foldr}, \textit{filter}, and \textit{range}

In combination, these functions are very powerful.

\textbf{Exercise}

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

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

\textbf{Hint}

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

```scheme
(define L (list 2 3 5 7 11))
(\(first\) L) (\(rest\) L)
\(\downarrow\) \(\downarrow\)
2 (list 3 5 7 11)
```

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

We want to go the other way:

We may use `cons` to construct lists. It consumes two values: `Any`, and a `(listof Any)`. It returns a list one longer, with the new value added at the left of the list.

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

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

Exercise

Construct \((\text{list } 6 \ 7 \ 42)\) using only \texttt{cons} and the empty list, \texttt{'}().

Exercise

Write a function \texttt{remove-second} that consumes a list of length at least two, and returns the same list with the second item removed.

\((\text{remove-second (list } 2 \ 4 \ 6 \ 0 \ 1)) \Rightarrow (\text{list } 2 \ 6 \ 0 \ 1)\)
Using foldr to construct lists

Recall what foldr does:
(foldr F base (list x0 x1 ... xn)) => (F x0 (F x1 (F ... (F xn base)))))

We can use foldr to copy a list.
(define (list-fun a b) (cons a b))

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

(foldr list-fun () (list 2 3 5))
(list-fun 2 (list-fun 3 (list-fun 5 '())))
=> (cons 2 (cons 3 (cons 5 '())))
=> (list 2 3 5)
Faking map

We can create new lists using `cons` and `foldr`, as if we were using `map`. Using `map`, I can add 2 to each value in a list:

```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))
```

Exercise

Write a function that consumes a `(listof Num)` and returns a 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`

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

**Exercise**

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

That is, it acts like `(filter even? L)`.

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

**Hint**

With `foldr` you have the “partial answer” from the previous call, which here must be a `(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 \ (\text{list } x_0 \ x_1 \ldots \ x_n)\) => \ (\text{list } (F \ x_0) \ (F \ x_1) \ldots \ (F \ x_n))
\]

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

**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.

\[
\text{(filter } G \ (\text{list } x_0 \ x_1 \ldots \ x_n)\) => \ (\text{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.
\]

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

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

This could be of any type, even a list.

\[
\text{(foldr } H \ \text{base} \ (\text{list } x_0 \ x_1 \ldots \ x_n)\) => \ (H \ x_0 \ (H \ x_1 \ (H \ldots \ (H \ x_n \ \text{base}))))
\]

\[
\text{(foldr } \ast \ 7 \ (\text{list } 2 \ 10 \ 3)) => \ 420
\]
If your function consumes a list, you may want to use one or more higher order functions. How to decide which one to use? Consider your **desired output**.

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

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

Recall what `foldr` does:

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

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.

**Exercise**

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

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

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

**Exercise**

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

Given that \texttt{use-foldr} consumes a \texttt{(listof Nat)}:
\begin{verbatim}
(define (use-foldr L) (foldr myfun "some-str" L))
\end{verbatim}
Since \texttt{foldr} is being used on a \texttt{(listof Nat)}, the first parameter of \texttt{myfun} must be \texttt{Nat}:
\begin{verbatim}
;; myfun: Nat \texttt{UnknownTypeX} -> \texttt{UnknownTypeX}
\end{verbatim}
Since the base is a \texttt{Str}, that must also be the type returned by \texttt{myfun}. So the whole contract is:
\begin{verbatim}
;; myfun: Nat \texttt{Str} -> \texttt{Str}
\end{verbatim}

I'm going to see what happens with this function:
\begin{verbatim}
(define (myfun n s) (string-append (number->string n) s))
(foldr myfun base L)
\end{verbatim}
What can we say about \texttt{base} and \texttt{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 \textit{Language} \rightarrow \textit{Choose language} \rightarrow \textit{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!
Tiny Functions with lambda

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

```scheme
;; (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.

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

```scheme
(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
```
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`!

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

Earlier we had the following functions:

```
(define (divisible? n d) (= 0 (remainder n d)))
(define (multiple-235? n)  
  (or (divisible? n 2) (divisible? n 3) (divisible? n 5)))
(define (keep-multiples-235 L) (filter multiple-235? L))
```

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

```
;; (keep-multiples 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.

;; (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 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)
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

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

Exercise
Use foldr to write a function \(\text{add-n-each} \ n \ \text{L}\) that adds \(n\) to each value in \(\text{L}\).

\((\text{add-n-each} \ 7 \ (\text{list} \ 2 \ 4 \ 8)) \Rightarrow (\text{list} \ 9 \ 11 \ 15)\)

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

\((\text{my-map} \ \text{sqr} \ (\text{list} \ 4 \ 5 \ 3)) \Rightarrow (\text{list} \ 16 \ 25 \ 9)\)
Faking filter with foldr using lambda

Exercise
Using `foldr`, write a function `(keep-multiples n L)` that returns the list containing all the values in L that are multiples of n.
That is, it acts like `(filter (lambda (x) (= 0 (remainder x n))) L)`.
`(keep-multiples 3 (list 1 2 3 4 5 6 7)) => (list 3 6)

Exercise
Use `foldr` to write a function that behaves like `filter`.
`(my-filter odd? (list 4 5 9 6)) => (list 5 9)`
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 $5 \times 5$:

\[
\begin{array}{c}
\text{list} & \text{list} & 1 & 2 & 3 & 4 & 5 \\
\text{list} & 2 & 4 & 6 & 8 & 10 \\
\text{list} & 3 & 6 & 9 & 12 & 15 \\
\text{list} & 4 & 8 & 12 & 16 & 20 \\
\text{list} & 5 & 10 & 15 & 20 & 25 \\
\end{array}
\]
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`.

**Exercise**
Write a function `(times-row n len)` that returns the nth 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))`.

`(times-row 3 4) => (list 3 6 9 12)`
`(times-row 5 3) => (list 5 10 15)`

**Hint**
Your function will be very simple, but you will need to use `lambda`!
Write a function \( \texttt{(times-table \ len)} \) that returns the \( n \times n \) times table. Use \texttt{times-row} as a helper function.

\[
(\text{times-table } 5) =>
(\textbf{list} (\textbf{list} 1 2 3 4 5) \\
(\textbf{list} 2 4 6 8 10) \\
(\textbf{list} 3 6 9 12 15) \\
(\textbf{list} 4 8 12 16 20) \\
(\textbf{list} 5 10 15 20 25))
\]
Mucking with adjacent values

Another thing we can do fairly (!) easily with foldr is making one value change the value that came after it in the list.

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

\begin{verbatim}
(define (F a b)
  (cond [(pred? a) ; When the "current" item is the kind we want...
         (cons (G (first b))) ; change the item *after* a.
        (rest b))]
       [else (cons a b)])) ; Otherwise, add new value at front.
\end{verbatim}
The function `muck-after-str` consumes a `(listof (anyof Num Str))`, and doubles all values in `L` that come immediately after any `Str`.

```
(define (muck-after-str L) (foldr ponder '() L))
```

Remember: any helper used with `foldr` has exactly two parameters.

**Exercise**
Write the helper function `(ponder new-item answer)` that allows `muck-after-str` to work.

```
(muck-after-str (list 2 7 "X" 3 5)) => (list 2 7 6 5)
```

**Exercise**
Change `ponder` so `muck-after-str` also changes every value that immediately follows the word "SQUARE" be the square of that number.

E.g.
```
(muck-after-str (list 5 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".

E.g.
```
(muck-after-str (list 5 7 "POP" 4 3)) => (list 5 7 3)
```

**Exercise**
Make the word "ADD" add up the two values that come after it.

```
(muck-after-str (list 5 7 "ADD" 7 3 5)) => (list 5 7 10 5)
```

Read about stacks, and be amazed.
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; transform(src.begin(), src.end(), dest.begin(), [](int i) { return i + 1; });</code></td>
</tr>
</tbody>
</table>
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)
  ```scheme```

- 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)
  ```scheme```
When to use `list` and when to use `cons`?

What’s the difference?

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

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

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

Except for creating examples, data, and other lists of known length, you should almost always use `cons` instead of `list`. 
Write a function `count-at` that consumes a `Str` and counts the number of times `\a` or `\t` appear in it.

\[\text{count-at}("A \text{ cat sat on a mat"}) \Rightarrow 7\]

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

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

\[(\text{acronymize} (\text{list} "Portable" "Network" "Graphics")) \Rightarrow "PNG"\]

\[(\text{acronymize} (\text{list} "GNU's" "Not" "UNIX")) \Rightarrow "GNU"\]
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.