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:

\[
\text{(define 2-multiples (range 0 1000 2))}
\]
\[
\text{(define 3-multiples (range 0 1000 3))}
\]
\[
\text{(define 5-multiples (range 0 1000 5))}
\]

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 \text{foldr} and \text{cond}, but it may be tricky. What to do?

I can check a single number easily enough:

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

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

\[
\text{(define (multiple-235? n)}
\]
\[
\text{\quad (or (divisible? n 2) (divisible? n 3) (divisible? n 5)))}
\]
Another higher order function: **filter**

**filter** consumes a predicate function `pred`, and `L`, which is a `(listof Any)`. `pred` must be a one-parameter function that consumes the type(s) of value in the list, and returns a `Bool`.

**filter** will return a list containing all the items `x` in `L` for which `(pred x)` returns `#true`.

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

For all values `x_k` for which `(F \ x_k) \Rightarrow \ #true`.

\[\text{;; keep-multiples-235:} \ (\text{listof Nat}) \rightarrow (\text{listof Nat})\]
\[\text{;; Example:}\]
\[(\text{check-expect} \ (\text{keep-multiples-235} \ (\text{range} \ 0 \ 10 \ 1)) \ (\text{list} \ 0 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 9))\]
\[(\text{check-expect} \ (\text{keep-multiples-235} \ (\text{list} \ 2 \ 4 \ 7 \ 7 \ 50 \ 4)) \ (\text{list} \ 2 \ 4 \ 50 \ 4))\]

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

\[(\text{define} \ (\text{keep-even} \ L) \ (\text{filter} \ \text{even?} \ L))\]
\[\text{;; (keep-even} \ (\text{list} \ 1 \ 2 \ 3 \ 4 \ 5 \ 6)) \Rightarrow \ (\text{list} \ 2 \ 4 \ 6)\]
filter practice

Here is an example of a function using \texttt{filter}:

\begin{verbatim}
(define (not-apple x) (not (equal? x "apple")))
(define (drop-apples L) (filter not-apple L))
\end{verbatim}

Exercise

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

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

Hint

You can use \texttt{first}, \texttt{second}, and \texttt{third} to get individual items from a list.
In combination, these functions are very powerful.

Exercise

Write a function \texttt{times-square} that consumes a (\texttt{listof Nat}) and returns the product of all the perfect squares in the list.

Hint: use \texttt{integer?} to check if a value is an integer.

\[(\text{times-square (list 1 36 5 4 1 7))} \Rightarrow (* 1 36 4 1) \Rightarrow 144\]
Handling extra parameters with lambda

Earlier we had the following functions:

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

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

```scheme
;; (keep-multiples n L) return all values in L which are divisible by n.
;; keep-multiples: Nat (listof Nat) -> (listof Nat)
;; Examples:
(check-expect (keep-multiples 7 (list 2 3 5 28 7 3 14 77)) (list 28 7 14 77))
```

I would like to use `filter`, but recall: the `Function` it consumes must have only one parameter.
Handling extra parameters with lambda

Solution: use lambda.

;; (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 \texttt{(listof Num)} and returns the list containing just the values which are greater than or equal to the average \texttt{(mean)} value in the list.
Constructing lists

Two functions which operate on lists, and which we will use more later, are **first** and **rest**:

\[
\text{(define L (list 2 3 5 7 11))}
\]
\[
\text{(first L) \quad (rest L)}
\]
\[
\downarrow \quad \downarrow
\]
\[
2 \quad \text{(list 3 5 7 11)}
\]

**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.

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

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

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

**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.

\((\text{remove-second } (\text{list } 2 \ 4 \ 6 \ 0 \ 1)) \Rightarrow (\text{list } 2 \ 6 \ 0 \ 1)\)
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 '())))
```

Exercise

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

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

```
(define (double-each L) (map (lambda (x) (+ x x))) L)
```
More faking map

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

**Exercise**

Use `foldr` to write a function `(add-1-each L)` that adds 1 to each value in `L`.

`(add-1-each (list 2 4 8)) => (list 3 5 9)`

**Exercise**

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

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

**Exercise**

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

`(my-map add1 (list 6 8 48)) => (list 7 9 49)`
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).
(keep-evens (list 1 2 3 4 5 6)) \Rightarrow (list 2 4 6)

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)) \Rightarrow (list 3 6)

Exercise

Use foldr to write a function that behaves like filter.
(my-filter odd? (list 4 5 9 6)) \Rightarrow (list 5 9)
Overview of Higher Order Functions

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

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

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

**filter**  Consider each item in a list, and returns a smaller list of the same items for which the predicate returns \texttt{#true}.

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

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

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

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

\[
\text{(foldr } * \ 7 \ (\text{list } 2 \ 10 \ 3)) \Rightarrow 420
\]
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>try <code>map</code></td>
</tr>
<tr>
<td>a list containing some of the items from the input</td>
<td>try <code>filter</code></td>
</tr>
<tr>
<td>a single value</td>
<td>try <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>try <code>foldr</code>, 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:
\[
\text{foldr } F \text{ base } (\text{list } x0 \ x1 \ldots \ xn) \implies (F \ x0 \ (F \ x1 \ (F \ldots \ (F \ xn \ \text{base})))))
\]

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

1. It says \((F \ x0 \ldots), (F \ x1 \ldots), \text{etc.}\)
   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 \((\text{listof Nat})\):
\[
\text{(define } \text{use-foldr } L \text{) (foldr } \text{myfun } \text{"some-str" } L))
\]

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 foldr

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

Since \textbf{foldr} is begin used on a (listof Nat), the first parameter of myfun must be \textbf{Nat}:
;; myfun: Nat UnknownTypeX -> UnknownTypeX

Since the base is a \textbf{Str}, that must also be the type returned by myfun. So the whole contract is:
;; myfun: Nat Str -> Str

I’m going to see what happens with this function:
(\textbf{define} (myfun n s) (string-append (number->string n) s))
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>vector&lt;int&gt; src = {2, 3, 5, 7, 11}, dest(5); transform(src.begin(), src.end(), dest.begin(), [](int i) { return i + 1; });</td>
</tr>
</tbody>
</table>
Str and higher order functions

A `(listof Str)` will work very well with higher order functions:

```
(foldr string-append "?" (list "hello" "how" "R" "U")) => "hellohowRU?"
```

```
(filter (lambda (s) (odd? (string-length s))) (list "first" "second" "third" "fourth")) => (list "first" "third")
```

**Exercise**

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.

```
(acronymize (list "Portable" "Network" "Graphics")) => "PNG"
(acronymize (list "GNU's" "Not" "UNIX")) => "GNU"
```
The character datatype **Char**

There is another datatype, **Char**, which represents a single character. Some **Char** values include `\a`, `\4`, `\space`, `\!`

Check equality using `char=?`

- `(char=? \e \e)` => **true**
- `(char=? \# \&)` => **false**

Char values can also be compared:

- `(char<? \e \q)` => **true**
- `(char<? \e \Q)` => **false**

...and ignoring uppercase/lowercase:

- `(char-ci=? \t \T)` => **true**
- `(char-ci<? \e \Q)` => **true**

Documentation for **Char** is with that for **Str**, on the course website.
Strings as lists of characters

In several ways, a Str resembles a list. Both have a length, and there are ways to get values from both:

- \( \text{length (list 6 42 7)} \Rightarrow 3 \)
- \( \text{second (list 6 42 7)} \Rightarrow 42 \)
- \( \text{string-length "hello world!"} \Rightarrow 12 \)
- \( \text{substring "hello world!" 1 2} \Rightarrow "e" \)

We can convert a Str to a (listof Char) and back, using string->list and list->string.

- \( \text{(string->list "hi there") \Rightarrow (list #\h #\i #\space #\t #\h #\e #\r #\e)} \)
- \( \text{(list->string (list #\h #\i #\space #\t #\h #\e #\r #\e)} \Rightarrow "hi there" \)

**Exercise**

Write a function drop-e that converts a Str to a (listof Char), replaces each #\e with a #\*, and converts it back to a Str.

- \( \text{(drop-e "hello world, how are you?")} \Rightarrow "h*ll* world, how ar* you?" \)
A subtlety of foldr

In general, the function consumed by foldr is non-commutative. That is, \((F \ a \ b)\) is not the same as \((F \ b \ a)\).

- The first parameter must be of the type of the values in the list.
- The second parameter must be of the same type as the returned value, and the base.

For example:

```scheme
;; (add-string-length s n) add n to the string-length of s
;; add-string-length: Str Nat --> Nat

(define (add-string-length s n)
  (+ n (string-length s)))

(foldr add-string-length 0 (list "a" "bb" "ccc"))
```

The base is 0, which is a \texttt{Nat}, so the second parameter and return value are also both \texttt{Nat}. The consumed list is a \texttt{(listof Str)}, so the first parameter must be a \texttt{Str}. 
Write a function `count-at` that consumes a `Str` and counts the number of times `\a` or `\t` appear in it.

```plaintext
count-at("A cat sat on a mat") => 7
```

Don’t use `filter`; complete this question using only `foldr`. 
When to use list and 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 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** in the wrong place, you’ll 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**.
Module Summary

- Use `filter` to select only certain values from lists.
- Combine `filter` with `map`, `range`, and `foldr`.
- Be able to use `lambda` to fill in extra parameters of helper functions when used with higher order functions.
- Use `cons` to construct lists. With `cons` and `foldr`, be able to manipulate lists without using `map` or `filter`.
- Start working with `Char`, by converting between a `Str` and a `(listof Char)`.

Before we begin the next module, please

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