If you have note already, please

- Read the Wikipedia entry on *Higher-order functions*. 
So far we have written only functions that consume one or a few values, and may combine them in various ways.

More often we have a collection of data to process.

Racket is a dialect of **LISP**, which was originally designed for **LIST Processing**.

Our principal way of grouping values is the **list**.
What is a list?

The word *list* comes from Old English “líste”, meaning a strip (such a strip of cloth or paper).

> “His targe wip gold list He carf atvo.”
> (Guy of Warwick, ca. 1330)

→ A strip of paper with items written on it.
→ An ordered collection of items.

We can make a list really easily. A few examples:

```lisp
(define wishes
  (list "comics" "turtle figures"
       "Donkey Kong" "Play-Doh Burger King"))

(define primes (list 2 3 5 7 11 13 17 19))
```
A value may be a list

Lists behave just like any other value.

We can define constants which are lists:

```
(define wishes
  (list "comics" "turtle figures"
       "Donkey Kong" "Play-Doh Burger King"))
```

```
(define primes (list 2 3 5 7 11 13 17 19))
```

We can have functions consume lists:

```
(length wishes) => 4
```

```
(first wishes) => "comics"
```

```
(rest wishes) => (list "turtle figures" "Donkey Kong" "Play-Doh Burger King")
```

We can have functions return lists:

```
(range 4 16 2) => (list 4 6 8 10 12 14)
```

```
(append (list 6 7 42) (list 3 5 15)) => (list 6 7 42 3 5 15)
```
In the design recipe, we specify the type of values in a list as follows:

- **Use** `(listof Type)` for a single type.
  
  *(listof Nat)* describes a list containing zero or more Nat. E.g. *(list 6 7 42)*
  
  *(listof Str)* describes a list containing zero or more Str. E.g. *(list "hello" "world")*

- **If a list may contain more than one type**, use `(listof (anyof Type1 Type2 ...))`.
  
  *(listof (anyof Num Str))* describes a list containing zero or more values, each of which is either a Num or a Str. E.g. *(list 3.14 "pie" "forty-two" -17)*

- **If a list is of known length and types**, use `(list Type1 Type2 ...)`.
  
  *(list Nat Str)* describes a list containing two values. The first value is a Nat, and the second value is a Str. E.g. *(list 6 "foo")*. 
  
  *(list "foo" 6)* is not a *(list Nat Str)*. It is a *(list Str Nat)*.
Lists and the design recipe

For each set of lists, find a type that describes all the lists. Try to be as specific as possible.
For example, (list 3 4 5) is a (listof Num), but it is also a (listof Int), and even more specifically a (listof Nat).

1. (list 4 3 -7), (list 3 1)
2. (list "We're" "all" "fine here, now," "thank" "you."), (list "How" "are" "you?")
3. (list "John" "Clark"), (list "Domingo" "Chavez"), (list "Dieter" "Weber")
4. (list 4 "*" 6 "=" 24), (list "sqrt" 4 "=" 2)
5. (list 2 4 5), (list)
6. (list (list 1 2) (list 3 4 5)), (list (list 6) (list -5 3))
Transforming items in a list using `map`

We can store data in a list, but what can we do with them?

There is a built-in function called `map` that transforms each item in a list, using a function.

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

It has two parameters: a Function and a `(listof Any)`. Some examples:

Exercise

Try out each expression with the given list, and one or two other lists. What happens?

1. \((\text{map } \text{sqr} \ (\text{list } 2\ 3\ 5))\)

2. \((\text{define} \ (\text{double-item } x) \ (*\ 2\ x))\)

   \((\text{define} \ (\text{double-each } L)\)

   \((\text{map} \ \text{double-item} \ L))\)

   \((\text{double-each} \ (\text{list} \ 0\ 1\ 2\ 3\ 4))\)
To use `map` on a list of values of some type:
write a function that consumes *one single value* of that type and transforms it as required.

For example, I wish to transform each item in a list by \( f(x) = 10\sqrt{x} \):

**Function to transform a single value:**

\[
\begin{align*}
\text{;; (10rootx n) return 10*sqrt(x)} \\
\text{;; 10rootx: Num -> Num} \\
\text{;; Requires: n >= 0} \\
\text{;; Examples:} \\
\text{(check-expect (10rootx 49) 70)}
\end{align*}
\]

```
(define (10rootx x) (* 10 (sqrt x)))
```

**Function to transform all items:**

\[
\begin{align*}
\text{;; (10rootx-each L) transform each item in L by 10rootx.} \\
\text{;; 10rootx-each: (listof Num) -> (listof Num)} \\
\text{;; Requires: each value is >= 0} \\
\text{;; Examples:} \\
\text{(check-expect (10rootx-each (list 49 81 100)) (list 70 90 100))}
\end{align*}
\]

```
(define (10rootx-each L) (map 10rootx L))
```
To use `map` on a list of values of some type:
write a function that consumes *one single value* of that type and transforms it as required.

**Exercise**

Digital signals are often recorded as values between 0 and 255, but we often prefer to work with numbers between 0 and 1.
Write a function `(squash-range L)` that consumes a `(listof Nat)`, and returns a `(listof Num)` so numbers on the interval \([0, 255]\) are scaled to the interval \([0, 1]\).

```
(squash-range (list 0 204 255)) => (list 0 0.8 1)
```

**Exercise**

Write a function that consumes a `(listof Str)`, where each `Str` is a person's name, and returns a list containing a greeting for each person.

```
(greet-each (list "Ali" "Carlos" "Sai")) => (list "Hi Ali!" "Hi Carlos!" "Hi Sai!")
```
Using `range` to build lists

`range` (start end step) returns the list that starts at `start`, and steps by `step` until just before it reaches `end`. This allows us to build new lists. Some examples:

- `(range 4 10 1) => (list 4 5 6 7 8 9)
- `(range 4 10 2) => (list 4 6 8)
- `(range 20 8 -3) => (list 20 17 14 11)
- `(range 20 8 3) => '() ;; the empty list

To work with `range` and `map`:

1. get proper values from `range`; test it.
2. use `map` to transform these values as needed.

Exercise

Complete the function `list-cubes`.

`; (list-cubes n) return the list of cubes from 1*1*1 to n*n*n.
`; list-cubes: Nat -> (listof Nat)
`; Examples:
(check-expect (list-cubes 4) (list 1 8 27 64))`
Summarizing a list using `foldr`

`range` lets us create a list, and `map` lets us transform each item. What if I want to my result to be a combination of the items in the list, instead of the entire list?

What is the total of all the values in \((\text{list} \ 6 \ 5 \ 8 \ 5 \ 14 \ 4)\)?

\[ (+ \ 6 \ (+ \ 5 \ (+ \ 8 \ (+ \ 5 \ (+ \ 14 \ 4)))) ) \] => 42

To do this automatically, there is another function, `foldr`, meaning “fold right”.

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

What does this mean?

We combine items, starting from the right, each time creating a new item to combine with.

\[
(\text{foldr} \ + \ 0 \ (\text{list} \ 6 \ 5 \ 8 \ 5 \ 14 \ 4))
\]

\[
\Rightarrow \ (+ \ 6 \ (+ \ 5 \ (+ \ 8 \ (+ \ 5 \ (+ \ 14 \ (+ \ 4 \ 0)))))))
\]

\[
\Rightarrow \ 42
\]
Strategy for working with foldr

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

1. Figure out what the answer is when the list is empty. Use this as the base.
2. Write a function that consumes two values, \texttt{new} and \texttt{old}, where \texttt{new} is a value from the list, and \texttt{old} is an answer.

For example: consider finding the sum of items in a (listof Num).

1. The sum of nothing is zero, so the base is \texttt{0}.
2. To calculate the sum of a value and another sum, just add the two values.

\[
\begin{align*}
(\text{define} \; (\text{add} \; a \; b) \; (+ \; a \; b)) \\
(\text{define} \; (\text{sum} \; L) \; (\text{foldr} \; \text{add} \; 0 \; L)) \\
(\text{sum} \; \langle \rangle) \Rightarrow 0 \\
(\text{sum} \; \langle \text{list} \; 5 \; 8 \; 4 \rangle) \Rightarrow (\text{add} \; 5 \; (\text{add} \; 8 \; (\text{add} \; 4 \; 0))) \Rightarrow 17
\end{align*}
\]

(We could use the built-in function \texttt{+}.)
Working with \texttt{foldr}

$$(\texttt{foldr} \ F \ \texttt{base} \ (\texttt{list} \ x_0 \ x_1 \ldots \ x_n)) \Rightarrow (F \ x_0 \ (F \ x_1 \ (F \ldots \ (F \ x_n \ \texttt{base}))))$$

1. Figure out what the answer is when the list is empty. Use this as the \texttt{base}. 
2. Write a function that consumes two values, \texttt{new} and \texttt{old}, where \texttt{new} is a value from the list, and \texttt{old} is an answer.

Exercise

Write a function \texttt{prod} that returns the product of a \texttt{(listof Num)}.

$$(\texttt{prod} \ (\texttt{list} \ 2 \ 2 \ 3 \ 5)) \Rightarrow 60$$

Exercise

Write a function \texttt{count-odd} that returns the number of odd numbers in a \texttt{(listof Nat)}.

Hint: read the documentation on \texttt{remainder}.

Can you do this using \texttt{map} and \texttt{foldr}? Just using \texttt{foldr}?
Exercises

Experiment with fold-sub.
Describe how it behaves, and why.
Write the contract and a better purpose statement.

;;; (fold-sub L) Do something mysterious with L.
;;; fold-sub: (listof Int) -> ...

(define (fold-sub L) (foldr - 0 L))
(fold-sub (list 6 5 2)) => ?

Read the documentation on string-length.
Write a function total-length that returns the total length of all the values in a (listof Str).

(total-length (list "hello" "how" "r" "u")) => 11
Write a function that returns the average (mean) of a non-empty (listof Num).

(check-expect (average (list 2 4 9)) 5)
(check-expect (average (list 4 5 6 6)) 5.25)

Recall that (length L) returns the number of values in L.

The factorial function, \( n! \), returns the product of the numbers from 1 to \( n \). For example, \( 4! = 1 \times 2 \times 3 \times 4 = 24 \).

Write a function (factorial n) that returns \( n! \).

(check-expect (factorial 5) 120)
(check-expect (factorial 1) 1)
Write a function \((\text{sum-square-difference } n)\) that consumes a \texttt{Nat} and returns the difference between the square of the sum of numbers from 0 to \(n\), and the sum of the squares of those numbers.

\[(\text{sum-square-difference } 3) \Rightarrow (- (\text{sqr } (+ 0 1 2 3)) (+ 0 1 4 9)) \Rightarrow 22\]
- Start storing information in lists, and describe lists in contracts.
- Transform list values using `map`, and `foldr`.
- Construct new lists using `range`, especially in combination with `map`.
- Use `foldr` to combine a list to a single value. This can be especially powerful when combined with `map`.
- Understand the use of `anyof` and be able to use it in your design recipes.