In Racket, select *Language → Choose language → Intermediate student with lambda.*

If you have not already, make sure you

- 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:

(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** \((\text{listof Type})\) for a single type.  
  \((\text{listof Nat})\) describes a list containing zero or more \(\text{Nat}\). E.g. \((\text{list } 6 7 42)\)  
  \((\text{listof Str})\) describes a list containing zero or more \(\text{Str}\). E.g. \((\text{list } "\text{hello}\" \"\text{world}\")\)

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

- If a list is of known length and types, use \((\text{list Type1 Type2 ...})\).  
  \((\text{list Nat Str})\) describes a list containing two values. The first value is a \(\text{Nat}\), and the second value is a \(\text{Str}\). E.g. \((\text{list } 6 \"\text{foo}\")\).  
  \((\text{list } \"\text{foo}\" 6)\) is not a \((\text{list Nat Str})\). It is a \((\text{list Str Nat})\).
Transforming items in a list using map

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

Use map to transform each item in a list, using a function.

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

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

$$(\text{define } \text{(double } x) \ (+ \ x \ x))$$

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

$$\text{(double-each } \text{(list } 0 \ 1 \ 2 \ 3 \ 4)) \Rightarrow \text{(list } 0 \ 2 \ 4 \ 6 \ 8)$$
Strategy for working with map

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.

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

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

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

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

(define (10rootx-each L) (map 10rootx L))
Working with map

To use \texttt{map} on a list of values of some type:
write a function that consumes \textit{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 \texttt{(squash-range L)} that consumes a \texttt{(listof Nat)}, and returns a \texttt{(listof Num)}
so numbers on the interval \([0, 255]\) are scaled to the interval \([0, 1]\).

\texttt{(squash-range (list 0 204 255))} => \texttt{(list 0 0.8 1)}
Using \texttt{range} to build lists

\texttt{(range start end step)} returns the list that starts at \texttt{start}, and steps by \texttt{step} until just before it reaches \texttt{end}. This allows us to build new lists.

\texttt{(range 4 10 1)} \Rightarrow \texttt{(list 4 5 6 7 8 9)}
\texttt{(range 4 10 2)} \Rightarrow \texttt{(list 4 6 8)}
\texttt{(range 20 8 -3)} \Rightarrow \texttt{(list 20 17 14 11)}
\texttt{(range 20 18 3)} \Rightarrow \texttt{()}; \texttt{the empty list}

To work with \texttt{range} and \texttt{map}:

1. get proper values from \texttt{range}; test it.
2. use \texttt{map} to transform these values as needed.

Exercise

Complete the function \texttt{list-cubes}.

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

\texttt{range} lets us create a list, and \texttt{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 (\texttt{list} 6 5 8 5 14 4)?

\((+ 6 (+ 5 (+ 8 (+ 5 (+ 14 4)))))) \Rightarrow 42\)

To do this automatically, there is another function, \texttt{foldr}, meaning “fold right”.

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

What does this mean?

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

\[
(\texttt{foldr } + \ 0 \ (\texttt{list} \ 6 \ 5 \ 8 \ 5 \ 14 \ 4))
\Rightarrow (+ 6 (+ 5 (+ 8 (+ 5 (+ 14 (+ 4 0))))) )
\Rightarrow 42
\]
Strategy for working with foldr

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

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, new and old, where new is a value from the list, and old is an answer.

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

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

\begin{verbatim}
(define (add a b) (+ a b))
(define (sum L) (foldr add 0 L))
(sum '()) => 0
(sum (list 5 8 4)) => (add 5 (add 8 (add 4 0))) => 17
(We could use the built-in function +.)
\end{verbatim}
Working with foldr

\[
\text{foldr } F \text{ base } (\text{list x0 x1 \ldots xn}) \rightarrow (F \ x0 \ (F \ x1 \ (F \ \ldots \ (F \ xn \ \text{base})))))
\]

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, new and old, where new is a value from the list, and old is an answer.

Exercise
Write a function \( \text{prod} \) that returns the product of a \( (\text{listof Num}) \).
\( \text{prod} \ (\text{list} \ 2 \ 2 \ 3 \ 5) \) \( \rightarrow \) 60

Exercise
Write a function that returns the number of odd numbers in a \( \text{listof Nat} \).
Hint: read the documentation on remainder.
Can you do this using \text{map} and \text{foldr}? Just using \text{foldr}?
Exercise

Experiment with `fold-sub`. Describe how it behaves, and why.

\[
\text{(define (fold-sub L) (foldr - 0 L))}
\]

`\text{(fold-sub (list 6 5 2)) \Rightarrow ?}`

Exercise

Write a function `flatten` that consumes a `(listof (listof Any))` and returns a list containing all the values in the lists.

\[
\text{(flatten (list (list 1 2) (list 3 4) (list 7))) \Rightarrow (list 1 2 3 4 7)}
\]

Hint: read the documentation on `append`.

Exercise

Read the documentation on `string-length`.

Write a function that returns the total length of all the values in a `(listof Str)`.
Exercise

Write a function that returns the average (mean) of a (listof Num).

(average (list 2 4 9)) => 5
(average (list 4 5 6 6) => 5.25)

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

Exercise

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 \( n! \).

(factorial 5) => 120
(factorial 1) => 1
Write a function \((\text{sum-square-difference } n)\) that consumes a \text{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.

\[
(\text{sum-square-difference } 3) \Rightarrow (- (\text{sqr} (+ 0 1 2 3)) (+ 0 1 4 9)) \Rightarrow 22
\]
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)

(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.

;; (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 function 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 helper functions, write a function that consumes a (listof Num) and returns a list containing the cube of each Num. (x³)
Handling extra parameters with lambda

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

```scheme
;; (add-5 n) add 5 to n.
;; add-5: Num -> Num
(define (add-5 n) (+ n 5))
```

```scheme
;; (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 a different 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 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.

\[(\text{define} \ \text{double} \ (\lambda \ (x) \ (* \ 2 \ x)))\]

\[(\text{double} \ 5) \Rightarrow 10\]

(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:

\[(\text{(lambda} \ (x \ y) \ (+ \ x \ y \ y)) \ 2 \ 5) \Rightarrow 12\]

Anything that can go in a function can go in a **lambda**, even another **lambda**:

\[(\text{(lambda} \ (x \ y) \ (\text{(lambda} \ (z) \ (+ \ x \ z)) \ y)) \ 4 \ 5)\]
Multi-argument map

So far we have used `map` only with functions that consume a single value: like `(map F L)`, where \(F\) is a single-parameter function and \(L\) is a list. But `map` can do so much more!

`map` works with any number of lists, all of the same length: `(map F L1 L2...)`

For example, we can implement vector addition very easily:

```scheme
;; A Vector is a (listof Num)

;; (vector-add a b) return the vector sum of a and b.
;; vector-add: Vector Vector -> Vector
;; Requires: a and b are of equal length.
;; Examples:
(check-expect (vector-add (list 2 3 3) (list 7 4 1)) (list 9 7 4))
```

**Exercise**

Write a function `(double-add a b)` that consumes two `Vector` and returns twice the vector sum of them.

`(double-add (list 2 3 3) (list 7 4 1)) => (list 18 14 8)`

Use `map` only once!
Recall that the distance of a point \((x, y)\) from \((0, 0)\), from the Pythagorean theorem, is

\[
\sqrt{x^2 + y^2}
\]

You may use the \texttt{sqrt} function to compute the square root. \((\texttt{sqrt 4}) \Rightarrow 2\).

\[
\underbrace{\text{Exercise}}_{\text{Write a function } (\texttt{distances xs ys}) \text{ that consumes two lists: the first contains } x \text{ values, and the second contains } y \text{ values. The output is a list containing the distance of each point } (x, y) \text{ from } (0, 0).}
\]

\((\texttt{distances (list 3 0 2) (list 4 7 2)}) \Rightarrow (\texttt{list 5 7 #i2.828427})\)
Multi-argument map

Here is one solution:

;; (distance x y) return the distance from (x, y) to the origin.
;; distance: Num Num -> Num
;; Example:
(check-expect (distance 3 4) 5)

(define (distance x y)
  (sqrt (+ (sqr x) (sqr y))))

;; (distances xs ys) return the list of distances for each of xs, ys.
;; distances: (listof Num) (listof Num) -> (listof Num)
;; Example:
(check-within (distances (list 3 0 2) (list 4 7 2))
  (list 5 7 2.8284) 0.0001)

(define (distances xs ys)
  (map distance xs ys))
Suppose we have two (listof Str): one of first names, and one of matching last names:

(define gnames (list "David" "James" "Douglas" "Burt" "Joseph"))
(define snames (list "Johnston" "Downey" "Wright" "Matthews" "Hagey"))

Exercise

Complete join-names.

;; (join-names G S) Make a list of full names from G and S.
;; join-names: (listof Str) (listof Str) -> (listof Str)
;; Example:
(check-expect (join-names gnames snames)
  (list "David Johnston" "James Downey" "Douglas Wright" "Burt Matthews" "Joseph Hagey"))
Module Summary

- 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`.
- Be able to use `lambda` functions in combination with `map` and `foldr`.
- Understand the use of `anyof` and be able to use it in your design recipes.

Before we begin the next module, please

- Read *How to Design Programs*, Section 4