CS115 - Module 9 - `filter`, `map`, and friends

Cameron Morland

Fall 2017

Reminder: if you have not already, ensure you:

- Read *How to Design Programs*, Intermezzo 3 (Section 18); Sections 19-23.

Abstraction

> *abstraction, n. 3a. . . . The process of isolating properties or characteristics common to a number of diverse objects, events, etc., without reference to the peculiar properties of particular examples or instances. (Oxford English Dictionary)*

We are going to look at certain classes of functions, and:

- find similarities (common properties)
- forget unimportant differences (properties of particular examples)
- consider how to deal with important differences

Compare these two functions:

```
;; (keep-even L) return all items in L
;;   that are even.
;; keep-even: (listof Nat)
;;                    -> (listof Nat)

(define (keep-even L)
  (cond [(empty? L) empty]
        [(even? (first L))
         (cons (first L)
               (keep-even (rest L)))]
        [else (keep-even (rest L))]))
```

```
;; (drop-apples L) return all items in L
;;   that are not 'apple.
;; keep-even: (listof Any)
;;                    -> (listof Any)

(define (drop-apples L)
  (cond [(empty? L) empty]
        [(not (equal? 'apple (first L)))
         (cons (first L)
               (drop-apples (rest L)))]
        [else (drop-apples (rest L))]))
```

Note the small difference. Could we somehow make a function that can replace both these functions, with the difference in a parameter?

Functions *are* values

Please use *Choose language* to switch to *Intermediate Student with Lambda*.

Now functions behave just like any other value. They can be bound to constants, put in lists or structures, consumed as arguments, and returned. Experiment with some examples:

```
(define the-constant-formerly-known-as-add +)
(the-constant-formerly-known-as-add 3 4)

;; (infix left-arg op right-arg) return the result of
;;   running op on left-arg and right-arg.
(define (infix left-arg op right-arg)
  (op left-arg right-arg))

(infix 5 - 4)
(infix 3 * 2)
(infix 3 * (infix 2 + 3))
```

A function to keep values described by a given function

```
;;  (keep-even L) return all items in L
;;    that are even.
;; keep-even: (listof Nat)
;;                    -> (listof Nat)

(define (keep-even L)
  (cond [(empty? L) empty]
        [(even? (first L))
         (cons (first L)
               (keep-even (rest L)))]
        [else (keep-even (rest L))]))
```

```
;;  (keep pred L) return all items in
;;    L for which pred returns true.
;; keep: Function (listof Any)
;;                          -> (listof Any)

(define (keep pred L)
  (cond [(empty? L) empty]
        [(pred (first L))
         (cons (first L)
               (keep pred (rest L)))]
        [else (keep pred (rest L))]))
```

`keep` is supremely awesome to use:

```
(define (keep-even L) (keep even? L))

(define (not-apple x) (not (equal? x 'apple)))
(define (drop-apples L) (keep not-apple L))
```

This pattern is so useful that there is a built-in function, **`filter`**, that behaves like `keep` !

`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`.

```
(define (keep-even L) (filter even? L))
;; (keep-even (list 1 2 3 4 5 6)) => (list 2 3 6)

(define (not-apple x) (not (equal? x 'apple)))
(define (drop-apples L) (filter not-apple L))
;; (drop-apples (list 'apple 4 'sale)) => (4 'sale)

(define (not-vowel? c)
  (not (member? c (list #\A #\E #\I #\O #\U #\a #\e #\i #\o #\u))))

(define (remove-vowels s)
  (list->string (filter not-vowel? (string->list s))))
;; (remove-vowels "filter is awesome!") => "fltr s wsm!"
```

`filter` practice

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

```
(define (not-apple x) (not (equal? x 'apple)))
(define (drop-apples L) (filter not-apple L))
```

Exercise: Use **`filter`** to write a function that keeps all multiples of 3.

```
(keep-multiples3 (list 1 2 3 4 5 6 7 8 9 10)) => (list 3 6 9)
```

Exercise: Use **`filter`** to write a function that keeps all multiples of 2 or 3.

```
(keep-multiples23 (list 1 2 3 4 5 6 7 8 9 10)) => (list 2 3 4 6 8 9 10)
```

Exercise: Use **`filter`** to write a function that keeps all items which are a Pythagorean triple $a < b < c : a^2 + b^2 = c^2$

```
(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)))
```

Tracing **`filter`** ?

We *could* trace **`filter`** by tediously working through the `keep` function we wrote earlier.

But we don't need to! Part of the point of abstraction is that it doesn't matter how **`filter`** works. **Somehow**, it takes all the items in the list that satisfy the predicate.

We abstract away the details of how it works. Does it do the left one first? The right one first? Something else? Who knows? It doesn't matter!

(Incidentally, the documentation for **`filter`** says "the `pred` procedure is applied to each element from first to last". So it *is* left-to-right, like our `keep` function.)

Another function that consumes a function

```
;; This is the sort from slide 42 of module 06, with pred added.
(define (insertion-sort pred alon)
  (cond [(empty? alon) empty]
        [ else (insert pred (first alon) (insertion-sort pred (rest alon)))]))

;; This is insert from slide 46 of module 06, with <= replaced by pred.
(define (insert pred n alon)
  (cond [(empty? alon) (cons n empty)]
        [(pred n (first alon)) (cons n alon)]; use pred instead of <=
        [ else (cons (first alon) (insert pred n (rest alon)))]))

(insertion-sort < (list 2 7 5 3 6)) => (list 2 3 5 6 7)
(insertion-sort > (list 2 7 5 3 6)) => (list 7 6 5 3 2)
(insertion-sort string<? (list "golf" "romeo" "echo" "alpha" "tango"))
=> (list "alpha" "echo" "golf" "romeo" "tango"))
```

Advanced: Implement `sort-students` from assignment 06 using this `insertion-sort` and your own predicate function.

Functional abstraction is awesome

If we create functions that consume functions, it can reduce code size. The difficult bits we can test really thoroughly (or have built-in to the language, where we hope they are well tested!). This makes our code more reliable!

Once we understand how a function like **`filter`** works we can read code that uses it. This is much easier than reading recursive code which might be slightly different from last time....

What is the cost? We need to expand our syntax a little: an expression may be a function, and the name of a function is a value.

Compare these two functions:

```
(define (X-upch c)
  (cond [(char-upper-case? c) #\-]
        [else c]))

;; (X-upper L) replace all uppercase
;;   letters in L with #\-
;; X-upper: (listof Char)
;;                    -> (listof Char)

(define (X-upper L)
  (cond [(empty? L) empty]
        [else
         (cons (X-upch (first L))
               (X-upper (rest L)))]))
```

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

;; (10rootX-each L) adjust each value
;;   in L using 10rootX.
;; 10rootX-each: (listof Num)
;;                    -> (listof Num)

(define (10rootX-each L)
  (cond [(empty? L) empty]
        [else
         (cons (10rootX (first L))
               (10rootX-each (rest L)))]))
```

These are very similar. How can we capture this sameness in a new function?

A function that modifies each value in a list using a given function

```
;; (X-upper L) replace all uppercase
;;   letters in L with #\-
;; X-upper: (listof Char)
;;                        -> (listof Char)

(define (X-upper L)
  (cond [(empty? L) empty]
        [else
         (cons (X-upch (first L))
               (X-upper (rest L)))]))
```

```
;; (xform f L) transform each item in
;;   L using f.
;; xform: Function (listof Any)
;;                        -> (listof Any)

(define (xform f L)
  (cond [(empty? L) empty]
        [else
         (cons (f (first L))
               (xform (rest L)))]))
```

`xform` is also supremely awsome to use:

```
(define (embiggen-each L) (xform add1 L))
;; (embiggen-each (list 2 3 5 7 11)) => (list 3 4 6 8 12)

(define (factorial n) (cond [(= n 0) 1] [else (* n (factorial (- n 1)))]))
(define (factorial-each L) (xform factorial L))
;; (factorial-each (list 2 3 5)) => (list 2 6 120)
```

This pattern is so useful that there is a built-in function, **`map`**, that behaves like `xform`

`map` consumes a function `func`, and `L`, which is a `(listof Any)`.
`func` must consume the type(s) of values in `L`. We will require that `func` be a one-parameter function.
There are no restrictions on the type that `func` returns.

```
(define (diminish-each L) (map sub1 L))
;; (diminish-each (list 2 3 5 7 11)) => (list 1 22 4 6 10)

(define (posn-distance p) (sqrt (+ (sqr (posn-x p)) (sqr (posn-y p)))))
(define (distance-each L) (map posn-distance L))
;; (distance-each (list (make-posn 5 12) (make-posn 3 4))) => (list 13 5)

(define (string-first s) (substring s 0 1))
(define (take-firsts L) (map string-first L))
;; (take-firsts (list "University" "of" "Waterloo")) => (list "U" "o" "W")
```

`map` practice

Here is an example of a function using **`map`**: ⟶

```
(define (string-first s) (substring s 0 1))
(define (take-firsts L) (map string-first L))
```

Exercise: Use **`map`** to write a function that doubles all items in a list.

```
(double-each (list 2 3 5)) => (list 4 6 10)
```

Exercise: Use **`map`** to write a function that gives the length of each string in a list.

```
(measure-lengths (list "hello" "how" "r" "u?")) => (list 5 3 1 2)
```

Exercise: Use **`map`** to write a function that consumes a number, n, and returns `'imaginary` if n is negative, and $\sqrt{n}$ otherwise.

```
(safe-sqrt (list 4 1 -1 -4)) => (list 2 1 'imaginary 'imaginary)
```

`map` can consume multiple lists

Although not offically part of this course, I want to show you one more handy feature of **`map`**.

It is not necessary that the function passed to **`map`** have only one parameters. The function passed to **`map`** must have as many parameters as you have lists.

Some examples:

```
(map + (list 11 13 17) (list 0 1 2)) => (list 11 14 19)}

(map substring
     (list "foo" "bar" "baz") (list 0 2 1)) => (list "foo" "r" "az")

(map max
     (list 42 6 7) (list 9 11 17) (list -100 7 13)) => (list 42 11 17)
```

Please *do not* use this feature on your assignments or exams. If there is a case where you would like to, there is likely some other technique that we want you to work with.

Counting up and down

Earlier we wrote recursive functions which followed the count-up template:

```
;; (countup-to n top) return the natural numbers from n up to top.
(check-expect (countup-to 2 6) (list 2 3 4 5))

(define (countup-to n top)
  (cond [(= n top) empty]
        [else (cons n (countup-to (add1 n) top))]))
```

I could modify this to instead put in some function of `n`. For example,

```
;; (countup-squares n top) return the squares of numbers from n up to top.
(define (countup-squares n top)
  (cond [(= n top) empty]
        [else (cons (sqr n) (countup-squares (add1 n) top))]))
```

I could do this with **`map`** and `countup-to`: `(map sqr (countup-to 0 4)) => (list 0 1 4 9)`

...but this sounds like something we might use a lot. There must be an easier way....

A way to create lists: `build-list`

It is common to create a sequence starting at zero, and going up by one. For example,

```
(first-n-squares 5) => (list 0 1 4 9 16)
```

For this Racket provides another built-in function,

`build-list: Nat Function -> (listof Any)`. The function must consume a `Nat`.

`build-list` behaves exactly the same as the following function:

```
(define (fake-build-lst n func) (map func (countup-to 0 n)))
```

So the `first-n-squares` function can be defined quite easily:

```
(define (first-n-squares n) (build-list n sqr))
```

Exercise: The nth triangular number is given by $T_n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$.
Use `build-list` to write a function that returns a list of the first n triangular numbers.
For example, `(make-triangles 5) => (list 0 1 3 6 10)`

Compare these two functions:

```
;; (sum L) return the sum of
;;   all items in L.
;; sum: (listof Num)
;;                -> Num

(define (sum L)
  (cond [(empty? L) 0]
        [else
         (+
          (first L)
          (sum (rest L)))]))
```

```
;; (join-words L) return all words
;;   in L, joined with spaces.
;; join-words: (listof Str)
;;                     -> Str

(define (join-words L)
  (cond [(empty? L) ""]
        [else
         (string-append " "
          (first L)
          (join-words (rest L)))]))
```

These both somehow combine `(first L)` with a recursive call on `(rest L)`. They both have a base case for **empty**. They both return a single value instead of constructing a list. How can we design a function to capture the common characteristics?

A function to join together all items

```
;; (sum L) return the sum of
;;   all items in L.
;; sum: (listof Num)
;;                  -> Num

(define (sum L)
  (cond [(empty? L) 0]
        [else
         (+
          (first L)
          (sum (rest L)))]))
```

```
;; (fold f base L) use f to join items
;;   in L with base.
;; fold: Function Any (listof Any)
;;                          -> Any

(define (fold f base L)
  (cond [(empty? L) base]
        [else
         (f
          (first L)
          (fold f base (rest L)))]))
```

Again, many recursive functions can be handled easily.

```
(define (sum L) (fold + 0 L))

(define (space-append w1 w2) (string-append " " w1 w2))
(define (join-words L) (fold space-append "" L))
```

Tracing `fold`

```
;; (fold f base L) use f to join items
;;   in L with base.
;; fold: Function Any (listof Any)
;;                        -> Any

(define (fold f base L)
  (cond [(empty? L) base]
        [else
         (f
          (first L)
          (fold f base (rest L)))]))
```

```
(fold + 2 (list 3 6 5))
⇒ (+ 3 (fold f 2 (list 6 5)))
⇒ (+ 3 (+ 6 (fold + 2 (list 5))))
⇒ (+ 3 (+ 6 (+ 5 (fold + 2 empty))))
⇒ (+ 3 (+ 6 (+ 5 2)))
⇒ (+ 3 (+ 6 7))
⇒ (+ 3 13)
⇒ 16
```

Another example: `(fold string-append "!" (list "hi" "there"))`

```
⇒ (string-append "hi" (fold string-append "!" (list "there")))
⇒ (string-append "hi" (string-append "there" (fold string-append "!" empty)))
⇒ (string-append "hi" (string-append "there" "!"))
⇒ (string-append "hi" "there!")
⇒ "hithere!"
```

The built-in function: **`foldr`**

There is a built-in function called **`foldr`** that does the same thing as our `fold`. **`foldr`** stands for “fold-right”, meaning it starts by combining the rightmost item with the base, then continues to the left.
(There is also `foldl`, “fold-left”, which starts by combining the leftmost item with the base, then continues to the right. It’s not officially part of this course, so please don’t use it in assignments or exams.)

`foldr` consumes a function `func`, a value `init`, which may be `Any`, and `L`, which is a `(listof Any)`. **`foldr`** combines the rightmost item in `L` and `init`, using `func`. It does this repeatedly, removing the last item in the list until it is empty.

```
(foldr F base (list x1 x2 ... xn)) => (F x1 (F x2 ... (F xn base) ... ))
```

```
(define (sum L) (foldr + 0 L))
;; (sum (list 5 7 11)) => 23

(define (longer-str w1 w2)
  (cond [(> (string-length w1) (string-length w2)) w1] [else w2]))
(define (longest-str L) (foldr longer-str "" L))
;; (longest-str (list "hi" "how" "r" "u?")) => "how"
```

Exercise: Use **`foldr`** and a helper function to complete `join-words`.

```
;; (join-words L) connect all items in L with a space before each.
;; join-words: (listof Str) -> Str
;; Example:
(check-expect (join-words (list "how" "r" "u?")) " how r u?")
```

A problem

Exercise: Use **`filter`** to complete `multiples-of`.

```
;; (multiples-of n L) return all items in L that are divisible by n.
;; multiples-of: Nat (listof Nat) -> (listof Nat)
;; Example:
(check-expect (multiples-of 3 (list 1 2 3 4 5 6 7 8 9 10)) (list 3 6 9))
```

Something like this sounds like it might work:

```
;; (is-mult-n? x) is x is divisible by n?
;; is-mult-n?: Nat -> Bool
(define (is-mult-n? x)
  (= 0 (remainder x n)))

(define(multiples-of n L)
  (filter is-mult-n? L))
```

...but when I run it, in `is-mult-n?` I get: *n: this variable in not defined*.

n is defined in `multiples-of`, but not in `is-mult-n?`. Is there a way to define a constant or function inside a function? Yes! It's called **`local`**, and will solve this problem.

`local`

All functions we have seen so far can be nested however we like — except **`define`** and `check-expect`. So far, we have only been able to use these at the top level, outside any expression.

The special form **`local`** lets us define things inside other things.

We can use **`local`** without defining anything. These two functions behave identically:

```
(define (my-fun-a x)
  (local [
          ]
    (+ x x)))

(define (my-fun-b x)
  (+ x x))
```

We may define things inside the **`local`** block.

```
(define (add-to-each n L)
  (local [
          ;; (add-n-to p) add p
          ;;    to the contant n.
          ;; add-n-to: Num Num -> Num
          (define (add-n-to p)
            (+ n p))
          ]
    (map add-n-to L)))
```

Using `local`

Exercise: Use **`local`** and **`filter`** to complete `multiples-of`.

```
;; (multiples-of n L) return all items in L that are divisible by n.
;; multiples-of: Nat (listof Nat) -> (listof Nat)
;; Example:
(check-expect (multiples-of 3 (list 1 2 3 4 5 6 7 8 9 10)) (list 3 6 9))
```

Here is a sample of how to use **`local`**:

```
(define (add-to-each n L)
  (local [
          ;; (add-n-to p) add p
          ;;     to the contant n.
          ;; add-n-to: Num Num -> Num
          (define (add-n-to p)
            (+ n p))
          ]
    (map add-n-to L)))
```

local and testing

It is impossible to independently test **local** definitions; `check-expect` does *not* work within local.
But if you don't test your helper functions, your program will be difficult to debug. So a suggestion: start by building your **local** functions as regular functions, and test them well.

```
;;  (add-n-to p) add p
;;      to the contant n.
;; add-n-to: Num Num -> Num
(define (add-n-to p)
  (+ n p))

;; Tests:
(define n 3) ; constant for add-n-to
(check-expect (add-n-to 5) 8)
(check-expect (add-n-to 1) 4)
```

After you test the function, remove the tests and move the function into the **local** block.

`local` and the design recipe

Include the **purpose**, **contract**, and any **requirements** for all local functions.

`local` constants

We are not required to define only functions inside `local`; we can define values as well. This can save enormous amounts of computation. Local can improve efficiency.

```
(define (remove-max-fast L)
  (local [(define biggest (foldr max (first L) L)) ; calculate biggest once
          (define (smallish n) (< n biggest)) ; compare to pre-computed
          ]
    (filter smallish L)))

(define (biggest X) (foldr max (first X) X))
(define (remove-max-slow L)
  (local [(define (smallish n) (< n (biggest L)))
          ]
    (filter smallish L)))

(time (first (remove-max-fast (build-list 10000 identity)))) ; takes 3 ms
(time (first (remove-max-slow (build-list 10000 identity)))) ; takes 4955 ms
```

A slow `list-max`

```
;; (list-max L) produces maximum in L.
;; list-max: (listof Num) -> Num
;; requires: L is nonempty
(define (list-max L)
  (cond
    [(empty? (rest L)) (first L)]
    [else
     (cond
       [(> (first L) (list-max (rest L))) (first L)]
       [else (list-max (rest L))])]))
```

Even for lists of length 22, (e.g. `(build-list 22 sqr)`), this takes many seconds. In fact, adding one more item can make it take twice as long.

This is because we compute `(list-max (rest L))` twice: once inside a question, and once inside an answer.

`local` is one way to fix this problem

```
;; (list-max2 L) produces maximum in L.
;; list-max2: (listof Num) -> Num
;; requires: L is nonempty
(define (list-max2 L)
  (cond
    [(empty? (rest L)) (first L)]
    [else
     (local [(define max-rest (list-max2 (rest L)))]
       (cond
         [(> (first L) max-rest) (first L)]
         [else max-rest]))]))
```

This function computes `(list-max2 (rest L))` just once, and saves it as a local variable. It will be reasonably fast.

`local` practice

Exercise: Using `local`, but without using recursion, write a function that consumes a `(listof Str)`, and returns a `(listof Str)` containing all the strings which are longer than the average length.

Exercise: Using `local`, but without using recursion, complete `keep-interval`.

```
;; (keep-interval low high L) keep values in L that are >= l and <= high.
;; keep-interval: Num Num (listof Num) -> (listof Num)
;; Example:
(check-expect (keep-interval 3 5 (list 3 1 6 5 4 2)) (list 3 5 4))
```

Returning lists using **`foldr`**

Recall that `(foldr func base L)` combines using `func` the rightmost item in `L` with base, then combines the next rightmost with that, and so on, until the list is reduced to one value.

```
(foldr F base (list x1 x2 ... xn)) => (F x1 (F x2 ... (F xn base) ... ))
```

...But a list is a value. So I can return it using **`foldr`**. I will rewrite `negate` using **`foldr`**:

```
;; (negate n) return -n.
;; negate: Num -> Num
(define (negate n) (- 0 n))

;; (negate-list L) negate each in L.
;; negate-list: (listof Num)
;;                   -> (listof Num)
(define (negate-list L) (map negate L))
```

```
;; (cons-negate a b) put -a before b.
;; cons-negate: Num (listof Num)
;;                   -> (listof Num)
(define (cons-negate a b)
  (cons (- 0 a) b))

;; (negate-foldr L) negate each in L.
;; negate-foldr: (listof Num)
;;                   -> (listof Num)
(define (negate-foldr L)
  (foldr cons-negate '() L))
```

Insertion sort using **foldr**

Recall the `insert` function we wrote earlier:

```
;; (insert item L) insert item into L so the result remains sorted.
;; insert: Num (listof Num) -> (listof Num)
;; Requires: L is sorted.
(define (insert item L)
  (cond [(empty? L) (cons item empty)]
        [(> item (first L)) (cons (first L) (insert item (rest L)))]
        [else (cons item L)]))
```

This function gives us almost everything we need to write insertion sort! Earlier we did the rest recursively, but we can use **foldr**:

```
;; (insertion-sort L) sort L, using insertion sort.
;; insertion-sort: (listof Num) -> (listof Num)
(define (insertion-sort L) (foldr insert '() L))

(insertion-sort (list 4 6 2))
    => (foldr insert '() (list 4 6 2))
    => (insert 4 (insert 6 (insert 2 '())))
    => (insert 4 (insert 6 (list 2)))
    => (insert 4 (list 2 6))
    => (list 2 4 6)
```

Re-using names with `local`

Recall that earlier we could re-use variable names:

```
(define x 1)
(define (foo x) (* 1 x))
(foo 2) => 2
```

At the top level, *x* is bound to 1. But then inside `foo`, when I run `(foo 2)`, *x* is bound to 2. A variable has the value to which it was *most recently* bound. (This may be slightly imprecise.)

What do you suppose `(foo 6)` returns?

```
(define x 1)

(define (foo x)
  (local [(define x 2)]
    (* 1 x)))

;; A: 1    B: 2    C: 3
```

It is possible to use **`local`** outside of a function. What do these programs do?

```
(local
  [(define x 1)
   (define y x)]
  (+ x y))
```

```
(local
  [(define x y)
   (define y 3)]
  (+ x y))
```

A few final notes about `local`

- `local` does not need to be the first thing inside a function.

```
(define (bar L)
  (cond [(empty? L) false]
        [else
         (local [(define item1 (first L))
                 (define the-rest (rest L))]
           (list item1 the-rest))]))
```

- There is nothing preventing you from using `local` inside another `local`.

```
(define (baz L)
  (cond [(empty? L) false]
        [else
         (local [(define item1 (first L))
                 (define the-rest (rest L))]
           (cond [(empty? the-rest) false]
                 [else
                  (local [(define item2 (first the-rest))
                          (define the-rest2 (rest the-rest))]
                    (list item1 item2 the-rest2))]))]))
```

That being said, it's probably a good thing to avoid. Usually it will be better to define a helper.

`lambda` is λ

Lisp (of which Racket is a dialect) was created to implement λ-calculus, a method of describing computation in mathematical terms. λ-calculus was created by Alonzo Church in the 1930s.

λ-calculus predates computers; it is the theory that allowed the design of computer hardware.

`lambda` and anonymous functions

In Racket, the `lambda` keyword lets us create functions without names: **anonymous functions**.

`(lambda (x) (+ x 5))` is a function with one parameter.

```
(map (lambda (x) (+ x 5)) (list 1 2 3)) => (list 5 6 7)
(filter (lambda (x)
          (or (even? x) (= 0 (remainder x 3))))
        (list 1 2 3 4 5 6 7 8 9 10)) => (list 2 3 4 6 8 9 10)
```

`lambda` functions are automatically local, and do not require any design recipe.

Exercise: Using `lambda` and `map`, but no helper functions, write a function that consumes a `(listof Num)` and returns a list with each item doubled.

Exercise: Using `lambda` and `filter`, but no helper functions, complete `multiples-of`.

```
;; (multiples-of n L) return all items in L that are divisible by n.
(check-expect (multiples-of 3 (list 1 2 3 4 5 6 7 8 9 10)) (list 3 6 9))
```

A few details about `lambda`

We can define named functions using **`lambda`** (though it's pointless):

```
(define double (lambda (x) (* 2 x)))
(double 5) => 10
```

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`**:

```
((lambda (v L)
   (filter (lambda (a) (> a v)) L))
 5 (list 3 4 5 6 7 8))
=> (list 6 7 8)

((lambda (a b)
   (cond [(= a b) 'equal]
         [(> a b) a]
         [else b]))
 5 5)
=> 'equal
```

`and` and `or` aren't functions...

Suppose I wanted to check if all items in a list are even. I might try:

```
;; (all-even? L) return true if all in L are true.
;; all-even?: (listof Bool) -> Bool
(define (all-even? L)
  (foldr and #true (map even? L)))
```

But that would **not work** because `and` is not a function — it is a special form.

One solution: I could create a wrapper function around `and`, then use that:

```
(define (all-even? L)
  (foldr (lambda (a b) (and a b)) #true (map even? L)))
```

This *will* work. But recall that `and` gives us **short-circuiting**.

So there is another function, `andmap`:

`(andmap pred L)` behaves like `(foldr (lambda (a b) (and a b)) #true (map pred L))`,

except it stops as soon as it finds a `#false`.

The function `ormap` works similarly, but for `or`.

Exercise: Write a function `(my-member? val L)` that returns `#true` if `val` is in `L`.

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:

Language	Code
Lisp, including Racket	`(map (lambda (x) (+ x 1)) (list 2 3 5 7 11))`
Python and Sage	`map(lambda x: x + 1, [2, 3, 5, 7, 11])`
Maple	`map(x -> x + 1, [2, 3, 5, 7, 11]);`
Haskell	`map (\x -> x + 1) [2, 3, 5, 7, 11]`
Javascript	`[2, 3, 5, 7, 11].map(function (x) { return x + 1; })`
Matlab and GNU Octave	`arrayfun(@(x) (x + 1), [2, 3, 5, 7, 11])`
Perl	`map { $_ + 1 } (2, 3, 5, 7, 11);`

C++

```
vector<int> src = {2, 3, 5, 7, 11}, dest(5);
transform(src.begin(), src.end(), dest.begin(),
          [](int i) { return i + 1; });
```

Practice

Do not use recursion for any of these exercises.

Exercise: Write a function `canadianize` that consumes a `Str` and returns the same `Str` with each `"o"` replaced by `"ou"`.

Exercise: Use **`foldr`** to write a function that returns the largest value in a `(listof Num)`.

Exercise: Write a function `sum-evens` that consumes a `listof Int` and returns the sum of all the even numbers in the list.

Exercise: Write a function `(sum-multiples L div)` that consumes two `(listof Nat)` and returns the sum of all values in `L` that are divisible by some value in `div`.

```
(sum-multiples (list 1 2 3 4 5) (list 2 3)) => 9
```

Module summary

Be comfortable reading and writing functions that use **`filter`**, **`map`**, `build-list`, and **`foldr`**.
Be able to use other functions that take functions as parameters.
See the benefits of **`local`** definitions: increased usefulness of abstract list functions, reduced repetition, improved readability, improved efficiency.
Read and write code that uses **`lambda`** to create anonymous local functions.
When global constants, local constants, and parameters share the same name, be able to determine which binding is relevant.

Before we begin the next module, please

- Read *How to Design Programs*, Sections 15 and 16.