```
#lang racket

;;**********************************************************************
;;*        Code reuse for list-processing functions                    *
;;**********************************************************************

;Warm-up: add-five

(define numbers (list 7 6 10 1))

(define (add-five l)
  (if (empty? l)
      l
      (cons (+ (first l) 5)
            (add-five (rest l)))))

(add-five numbers)

What if instead, we wanted to add 7 to every item in the list?
We could write a new function called addSeven:

(define (add-seven l)
  (if (empty? l)
      l
      (cons (+ (first l) 7)
            (add-seven (rest l)))))

(add-seven numbers)


Instead, we can add another argument

(define (add-number n l)
  (if (empty? l)
      l
      (cons (+ (first l) n)
            (add-number (rest l)))))

Does that look right?

(define (add-number-fixed n l)
  (if (empty? l)
      l
      (cons (+ (first l) n)
            (add-number-fixed n (rest l)))))

```

Now we can re-define add-seven and add-five with add-numbers-fixed:

```
(define (add-five2 l)
  (add-number-fixed 5 l))

(define (add-seven2 l)
  (add-number-fixed 7 l))

(add-five2 numbers)

(add-seven2 numbers)
```

But wait. Here's another function that takes a list as its argument:

```
(string-append "lions" "!")

(define (exclaim l)
  (if (empty? l)
      l
      (cons (string-append (first l) "!")
            (exclaim (rest l)))))
```

exclaim takes a list of strings as input and produces a new list of strings as output, where each string in the output list is a string in the input list with an exclamation mark added to it.

```
(define animals (list "cat" "bird" "pig" "raccoon"))

(exclaim animals)

animals
```

Now we have three functions that each take a single list as an argument: addFiv addSeven, and exclaim.
What do these functions have in common?

[Answer: each takes a list and does something to each item in the list.]

This is a common programming pattern: take a list and an operation and apply th operation to each item in the list, in order to create a new list.

What do I mean by 'apply an operation'?

In the case of addFive, we can separate it into two parts.
There is a little helper function that does the actual addition, and the body o
the function, which applies the helper to each item in the list and produces
the new list.

```racket
(define (plus-five x) (+ x 5))

(define (add-five3 l)
  (if (empty? l)
      l
      (cons (plus-five (first l))
            (add-five3 (rest l)))))

(add-five3 numbers)
```

Similarly, we can separate addSeven and exclaim into two parts:

```racket
(define (plus-seven x) (+ x 7))

(define (add-seven3 l)
  (if (empty? l)
      l
      (cons (plus-seven (first l))
            (add-seven3 (rest l)))))

(add-seven3 numbers)

(define (add-exclamation w) (string-append w "!"))

(define (exclaim2 l)
  (if (empty? l)
      l
      (cons (add-exclamation (first l))
            (exclaim2 (rest l)))))

(exclaim2 animals)
```

Well done! We've taken our three original functions and made six functions.
So much more compact!

The thing is, once we take out the helper function, these three functions
look remarkably similar:

In fact, the only difference among them is what helper function they apply to the items in the list.
If only there was some general-purpose function to package up the shared parts of each of these functions, the part that applies the helper function and returns a list.

```
;;****************************************************************************
;;*                 Introducing map                                          *
;;****************************************************************************
```

This operation is called MAP.
Map is a built-in function in Racket which takes a list and a function as its arguments, and applies the function to each item in the list, returning a new list.

We can rewrite addFive with map as follows:

```
(define (add-five-map l) (map plus-five l))

(add-five-map numbers)
```

See how elegant that is! Map takes care of the recursion through the list and the creation of the new list for us.

We can rewrite addSeven and exclaim similarly.

```
(define (add-seven-map l) (map plus-seven l))

(add-seven-map numbers)

(define (exclaim-map l) (map add-exclamation l))

(exclaim-map animals)
```

```
;;****************************************************************************
;;*                 Higher-order functions                                   *
;;****************************************************************************
```

What makes map so powerful is that it takes a function as an argument.

This makes it a *higher-order function.*

A higher-order function is simply a function that takes another function as an argument.

We can write our own higher-order functions as well as using built-in higher order functions like map.
For instance, we can write a function that takes in a function and applies it to the string "cat".

```
(define (apply-to-cat f)
  (f "cat"))
```

apply-to-cat takes any function that takes a single string as an argument, and applies that function to the string "cat."

Let's try it out with our addExclamation function.

```
(apply-to-cat add-exclamation)
```

```
;; ***********************************************************************
;; *                  Implementing map                                   *
;; ***********************************************************************
```

What is map really doing behind the scenes?
It's pretty much identical to the code we had in our very first addFive and exclaim functions.

Map takes a function and list and applies the function recursively to the items in the list.

If the list is empty, it returns the empty list.

Otherwise, it applies the function to the first item in the list and uses cons to create a new list consisting of the result and map applied to the rest of the list.

```
(define (my-map f l)
  (if (empty? l)
      l
      (cons (f (first l))
            (my-map f (rest l)))))

(my-map (lambda (x) (string-append x "!")) animals)
```

```
;; ***************************************************************************
;; *                    Lambdas again                                        *
;; ***************************************************************************

Why are anonymous functions useful? Let's return to map.

Instead of having to define the helper functions plusFive and
addExclamation outside of the main functions addFive and exclaim, we can write
them as anonymous functions in the arguments to map.

(define (add-five4 l)
  (map (lambda (x)(+ x 5)) l))

(add-five4 numbers)

(define (exclaim3 l)
  (map (lambda (x)(string-append x "!")) l))

(exclaim3 animals)

;; ****************************************************************************
;; *                          Properties of map                               *
;; ****************************************************************************

Map is one example of a higher-order function: it is a function that takes
a list and another function, and applies the function to each element in the
list, returning a list of the results.

Now that we've seen how to use map and how it is implemented, let's talk about
some properties of map.

;;*****************************************************************************
;;*       Property 1: Return list doesn't need to share input list type       *
;;*****************************************************************************

This map example shows another property of map: the lists it returns don't have
have the same type of values as the list it takes as input.
In the example above, the input list contained functions, but the output list
contained strings.

Instead of adding 5 to every item in a list, for instance, perhaps we want to k
whether each item in the list is divisible by 5.
```

```racket
First, how can we check whether a single  number is divisible by 5?

 [Answer: using modulo]

Racket has a built-in modulo function that we can use:

(modulo 10 5)

To check whether 10 is evenly divisible by 5, we check whether the output of
calling modulo is zero.

(= (modulo 10 5) 0)

How do we package up these operations into a function to pass to map?

Work with your neighbor to write a function that checks whether a given number
is evenly divisible by 5.

(define (is-divisible-by-five n) (= (modulo n 5) 0))

Now we want to map this function over our list to check whether each element is
evenly divisible by five.

We can either map this named function over the list, or give it to the list as
an anonymous function:

(define (mod-5-list1 l) (map is-divisible-by-five l))
(define (mod-5-list2 l) (map (lambda (x) (= (modulo x 5) 0)) l))

(mod-5-list1 numbers)
(mod-5-list2 numbers)

Exercise: make this even more flexible by writing a function that takes a numbe
Boolean values indicating whether each item in the list is divisible by that nu

(define (is-divisible l n) (map (lambda (x) (= (modulo x n) 0)) l))

(is-divisible numbers 2)
(is-divisible numbers 7)
```