

---

# Higher Order Functions Activity

Mattox Beckman

---

## Why

Mapping, folding, and zipping allow us to abstract away common list computations. Knowing how to use them will make you more productive as a programmer.

## Learning Objectives

1. Discover how to reduce code by using `map` and `reduce`.
2. Discover how to take a fix-point.
3. Transform the interface of a function using `curry`, `uncurry`, and `flip`.

## Mapping and Folding

1. Use `map` to write a function that negates the elements of a list. Here is the recursive version.

```
1 (defn neg-list [x]
2   (if (empty? x) empty
3       (cons (- (first x))
4             (neg-list (rest x)))))
```

2. Now try writing your own version of `map`.

3. Oh, wait, you can also do stuff like this:

```
1 user> (map + '(10 20 30) '(40 50 60))
2 '(50 70 90)
```

Do you think you can write that?

## Other HOFs

4. The fix-point of function  $f$  is a value  $x$  such that  $f(x) = x$ . Write a function `fix` that takes a function  $f$  and returns its fix-point.

```
1 user> (cos 1)
2 0.5403023058681398
3 user> (cos (cos 1))
4 0.8575532158463934
5 user> (fix cos 1)
6 0.7390851332151607
```

## Functions as Data

5. Write a function `flip` that takes a function that takes two arguments and returns an equivalent function where the arguments have been reversed.

```
1 user> (defn sub [a b] (- a b))
2 user> ((flip sub) 10 2)
3 -8
```

6. Write a function `todo` that takes a list of *functions* and applies them in turn to a “victim.”

```
1 user> (todo (list inc double) 20)
2 42
```