

# Higher Order Functions

*September 20, 2018*

---

# Warm-up: list processing

---

Write a function that takes a list and adds 5 to each item in the list.

---

# Code reuse

---

What if we want to add 7 instead of 5?

---

# Code reuse

---

Why is it a bad idea to copy code?

<pre>(define (add-five l)   (if (empty? l)       1       (cons (+ (first l) 5)             (add-five (rest l))))))</pre>	<pre>(define (add-seven l)   (if (empty? l)       1       (cons (+ (first l) 7)             (add-seven (rest l))))))</pre>
--------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------

---

# Map

---

Map is a function that takes a list and a function as its arguments, and applies the function to each item in the list, returning a new list.

```
> (map (lambda (x) (+ 5 x)) (list 1 2 3))  
'(6 7 8)
```

---

# Higher-Order Functions

---

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

---

# Defining map

---

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

---

# Functions are first-class citizens

---

- ❖ Do not need to be named (lambdas)
- ❖ Can be returned by functions
- ❖ Can be arguments to functions



---

# Anonymous functions revisited

---

Anonymous functions are useful when we want to feed a function into a higher-order function like `map`, and we don't care about being able to reference it later.

---

# Recap

---

**First-class functions:** functions that are treated just like other values in the language, including being able to appear in all syntactic environments.

**Higher-order functions:** functions that take functions as arguments.

---

# Properties of Map

---

- ❖ Input items and return items do not need to be of the same type
- ❖ Preserves the length of the original list

---

## Exercise: write a generic isDivisible function

---

Using map, write a function that takes a number and a list, and returns a list of Boolean values indicating whether each item in the list is divisible by that number.

```
> (is-divisible 4 (list 14 16 20))  
  '(#f #t #t )
```

---

# Filter

---

Another useful higher-order function is filter, which filters out items from the list based on the function supplied.

```
> (filter (lambda (x) (> x 5)) (list 5 6 7))  
‘(6 7)
```

---

# Properties of Filter

---

- ❖ Function given as argument must return a boolean
- ❖ Does not preserve the length of list
- ❖ Returns copies of items from the original list

---

# Filter exercise

---

Use `filter` to write `all-titlecase`, a function that filters out strings that are not in title-case.

Hint: you may use the built-in `string-titlecase` function, which returns a copy of a string in titlecase.

```
> (all-titlecase (list "Cat" "cat" "CAT"))  
('("Cat"))
```

---

# Bonus Map property: function composition

---

The result of mapping two functions over a list is the same as mapping the composition of the two functions over the list.

$$(\text{map } f2 (\text{map } f1 \text{ lst})) == (\text{map } f1 \oplus f2 \text{ lst})$$

$$(\text{map } \text{add5} (\text{map } \text{add5} \text{ lst})) == (\text{map } \text{add5} \oplus \text{add5} \text{ lst})$$

$$(\text{map } \text{add5} (\text{map } \text{add5} \text{ lst})) == (\text{map } \text{add10} \text{ lst})$$