Higher-order functions
Write copy for a list L

; test
(check-expect (copy L) L)

(define (copy L) L)

You can’t say this ↑. Use recursion :).
(define (copy L)
  (if (empty? L)
    '()
    (cons (first L) (copy (rest L))))

More Racket practice
See if you can spot some patterns & start to develop good techniques for Racket
(define (plus-one L)
  (if (empty? L)
      '()
      (cons (+ 1 (first L)) (plus-one (rest L))))))
(define (one-pluser n) (+ n 1))

(define (plus-one L)
  (if (empty? L)
      '()
      (cons (one-pluser (first L)) (plus-one (rest L))))

\[
\begin{array}{c}
\text{1} \rightarrow 2 \rightarrow 3 \\
\downarrow \quad \downarrow \quad \downarrow \\
\text{2} \rightarrow 3 \rightarrow 4 \\
\downarrow \quad \downarrow \\
\text{...} \rightarrow 9 \\
\downarrow \quad \downarrow \quad \downarrow \\
\text{...} \rightarrow 10
\end{array}
\]
(define (doubler n) (* n 2))

(define (double L)
  (if (empty? L)
      '()
      (cons (doubler (first L)) (double (rest L)))))
A common pattern

Apply a transformer function to each element of a list

\[
\text{(define (one-pluser n) (+ n 1))}
\]
\[
\text{(define (plus-one L)}
\]
\[
\text{  (if (empty? L)}
\]
\[
\text{    '())}
\]
\[
\text{  (cons (one-pluser (first L)) (plus-one (rest L))))}
\]
\[
\text{(define (doubler n) (* n 2))}
\]
\[
\text{(define (double L)}
\]
\[
\text{  (if (empty? L)}
\]
\[
\text{    '())}
\]
\[
\text{  (cons (doubler (first L)) (double (rest L))))}
\]

1 → 2 → 3

... → 9

1 → 2 → 3

... → 10

1 → 2 → 3

... → 9

2 → 3 → 4

... → 10

2 → 4 → 6

... → 18
Higher-order functions
(functions as values)
A common pattern
Apply a transformer function to each element of a list

\[
\text{(define (one-pluser n) (+ n 1))}
\]
\[
\text{(define (plus-one L)}
\]
\[
\text{  (if (empty? L)}
\]
\[
\text{    '())}
\]
\[
\text{      (cons (one-pluser (first L)) (plus-one (rest L)))))}
\]

\[
\text{(define (doubler n) (* n 2))}
\]
\[
\text{(define (double L)}
\]
\[
\text{  (if (empty? L)}
\]
\[
\text{    '())}
\]
\[
\text{      (cons (doubler (first L)) (double (rest L)))))}
\]

\[
\begin{array}{c}
\text{1} \rightarrow 2 \rightarrow 3 \rightarrow \cdots \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow \cdots \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow \cdots \rightarrow 5 \rightarrow 10 \rightarrow 9 \rightarrow 18
\end{array}
\]
(define (one-pluser n) (+ n 1))

(define (plus-one L) (map one-pluser L))

(define (doubler n) (* n 2))

(define (double L) (map doubler L))
(define (odd?  n) (= (modulo n 2) 1))

(define (odds  L)
  (cond [(empty? L) '()] [(odd? (first L)) (cons (first L) (odds (rest L)))] [else (odds (rest L))])))
(define (old? n) (> n 30))

(define (olds L)
  (cond [(empty? L) '()] [(old? (first L)) (cons (first L) (olds (rest L)))] [else (olds (rest L))])))
A common pattern
Use a predicate function to cull a list

(define (odd? n) (= (modulo n 2) 1))

(define (odds L)
  (cond [[(empty? L) '()]
        [(odd? (first L)) (cons (first L) (odds (rest L)))]
        [else (odds (rest L))])))

(define (old? n) (> n 30))

(define (olds L)
  (cond [[(empty? L) '()]
        [(old? (first L)) (cons (first L) (olds (rest L)))]
        [else (olds (rest L))])))
(define (odd? n) (= (modulo n 2) 1))

(define (odds L) (filter odd? L))

(define (old? n) (> n 30))

(define (olds L) (filter old? L))
(define (sum L)
  (if (empty? L)
      0
      (+ (first L) (sum (rest L)))))

1 + 2 + 3
1 + 9 + 0 = 45
(define (product L)
  (if (empty? L)
      1
      (* (first L) (product (rest L)))))
A common pattern
Reduce a list to a value by accumulating the list's elements

```
(define (sum L)
  (if (empty? L)
      0
      (+ (first L) (sum (rest L)))))
```

```
(define (product L)
  (if (empty? L)
      1
      (* (first L) (product (rest L)))))
```

```
1 → 2 → 3 → \ldots
1 + 2 + 3 = 6
```

```
1 → 2 → 3 → \ldots
1 \times 2 \times 3 = 6
```

```
1 \ldots 9
1 + 2 + 3 + \ldots + 9 + 0 = 45
```

```
1 \ldots 9
1 \times 2 \times 3 \times \ldots \times 9 \times 1 = 362880
```
(define (sum L)  
  (foldr + 0 L))

(define (product L)  
  (foldr * 1 L))
\[ \text{(define (sum L)} \]
\[ \text{\quad (foldr + 0 L))} \]

\[ \text{foldr} \]
Reduce a list to a value by accumulating the list’s elements (start from the right)

\[ \text{(define (product L)} \]
\[ \text{\quad (foldr \ast 1 L))} \]

watch out for commutativity!
\[ (\text{foldr } f \text{ seed } L) \equiv (f \, v_0 \ldots (f \, v_n \text{ seed}) \ldots) \]
A common pattern

Define one function to be used as an argument to another function

\[
\text{(define (one-pluser n) (+ n 1))}
\]

\[
\text{(define (plus-one L) (map one-pluser L))}
\]
Anonymous functions ($\lambda$)

Define one function to be used as an argument to another function

```scheme
(define (plus-one L)
  (map (lambda (x) (+ x 1)) L))
```
Anonymous functions (\(\lambda\))

\[
(\text{define } \text{one-pluser } x) \ (\text{+ } x \ 1))
\]

\[
(\text{define one-pluser (lambda } (x) \ (\text{+ } x \ 1)))
\]
Implement these functions

Using `map`, `filter`, `foldl`, and `foldr`

**times3**: multiplies each number in a list $L$ by 3.
*you can assume $L$ contains only integers*

**9multiples**: filters a list $L$ to remove any numbers that aren’t divisible by 9.
*you can assume $L$ contains only integers*

**sum-of-times10**: multiplies each element of a list $L$ by 10, then sums the results
*you can assume $L$ contains only integers*

**count-ones**: counts the number of times the value 1 appears in a list $L$
*you can assume $L$ contains only integers*

*write one version using a combination of `filter` and `foldr`*
*write another version using only `foldr`

**BONUS**

*copy, reverse*

*with `map`? with `foldl` or `foldr`?*

*map, filter, foldr, foldl*

**Practice test-driven-development (TDD).**
- write all your own tests!

**You can practice pair programming.**
- switch off after every function
- increase your understanding
MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution