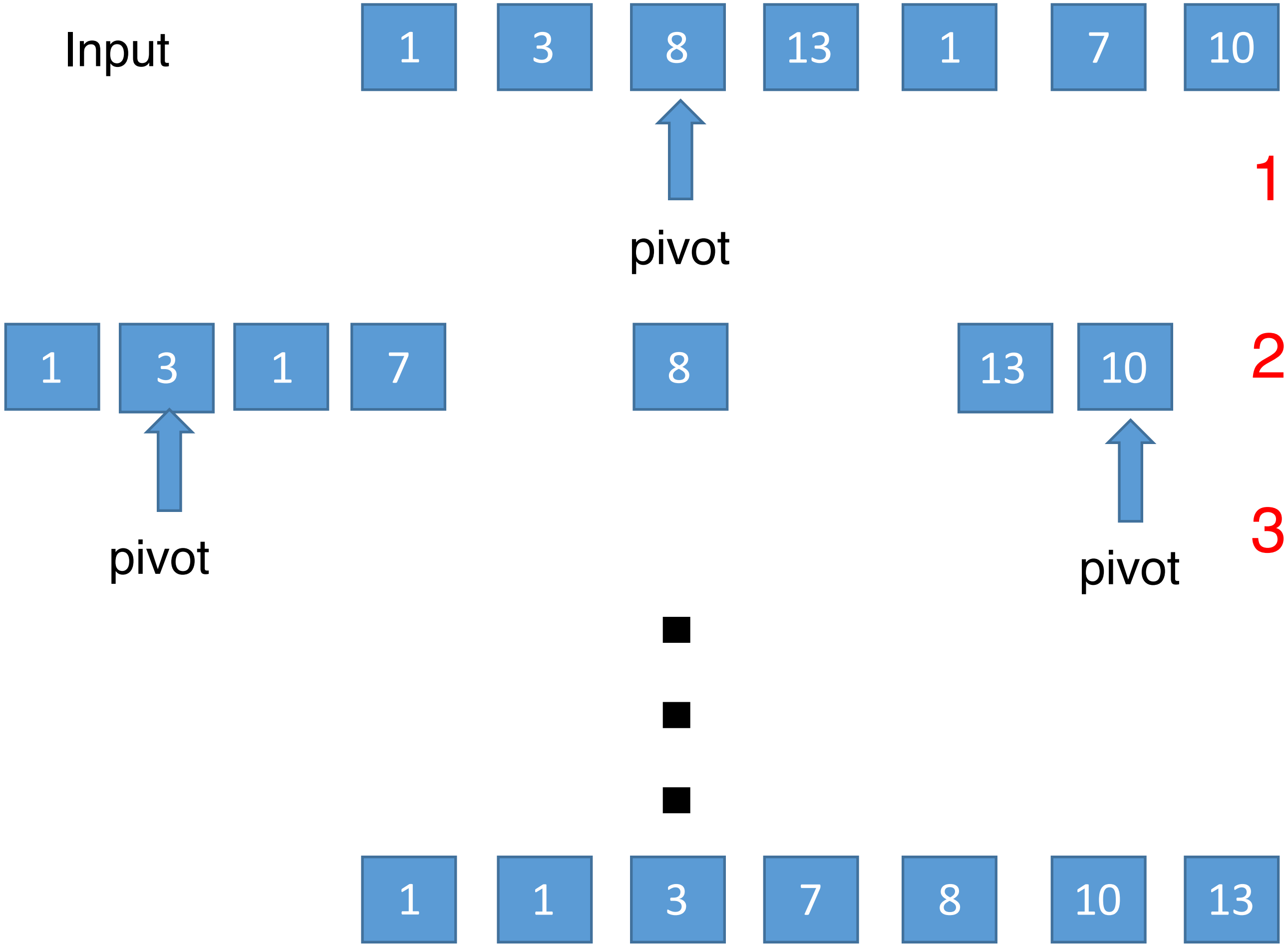
Functional Programming and Scheme

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Quick Sort



Input

1 3 8 13 1 7 10

pivot

1 3 1 7

8

13 10

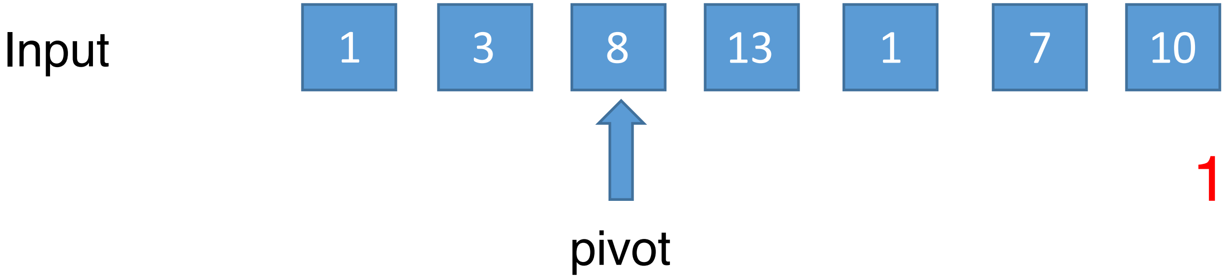
pivot

pivot

1 1 3 7 8 10 13

1. Find “pivot”
2. Split list
3. Recursion

Quick Sort



Input

1 3 8 13 1 7 10

pivot

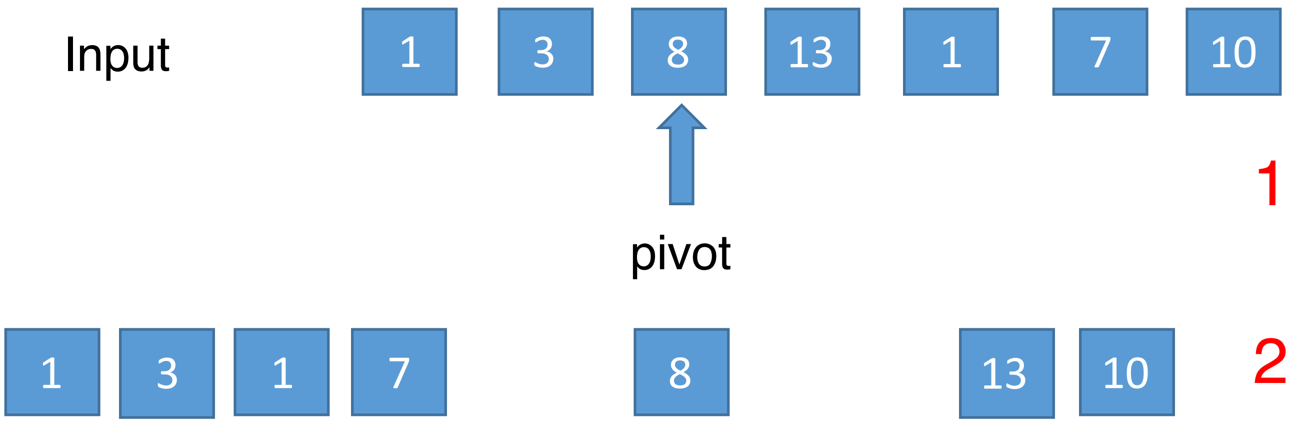
1. Find “pivot”

```
(define (getPivot lst) (list-ref lst (random (length lst))))
```

Get elem. at a position

Get random position

Quick Sort



1. Find “pivot”

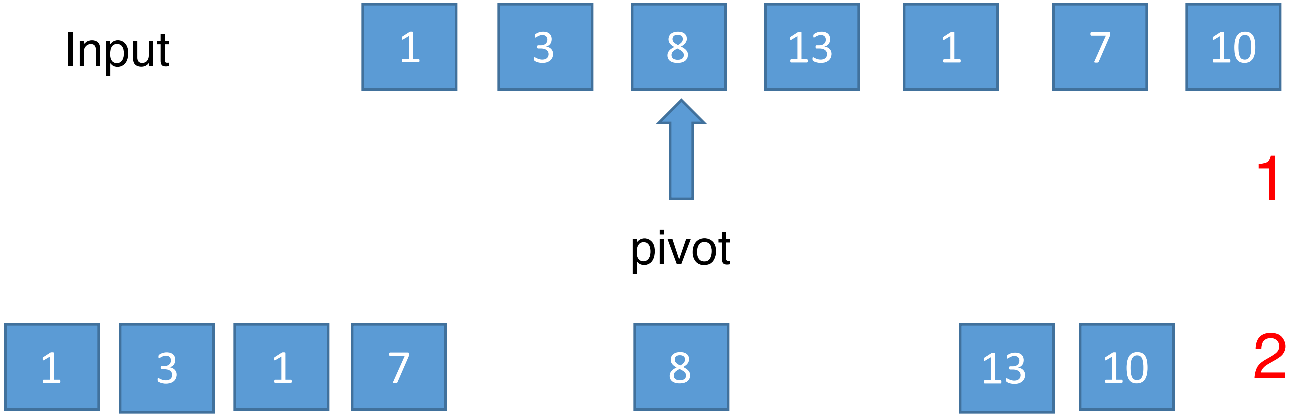
2. Split list

```
(filter (> pivot) lst)

(filter (= pivot) lst)

(filter (< pivot) lst)
```

? ✗

<, =, > take two parameters

Quick Sort



Input

1 3 8 13 1 7 10

pivot

1 3 1 7

8

13 10

1. Find “pivot”

2. Split list

```
(filter (lambda (x) (> pivot x)) lst)

(filter (lambda (x) (= pivot x)) lst)

(filter (lambda (x) (< pivot x)) lst)
```

Currying

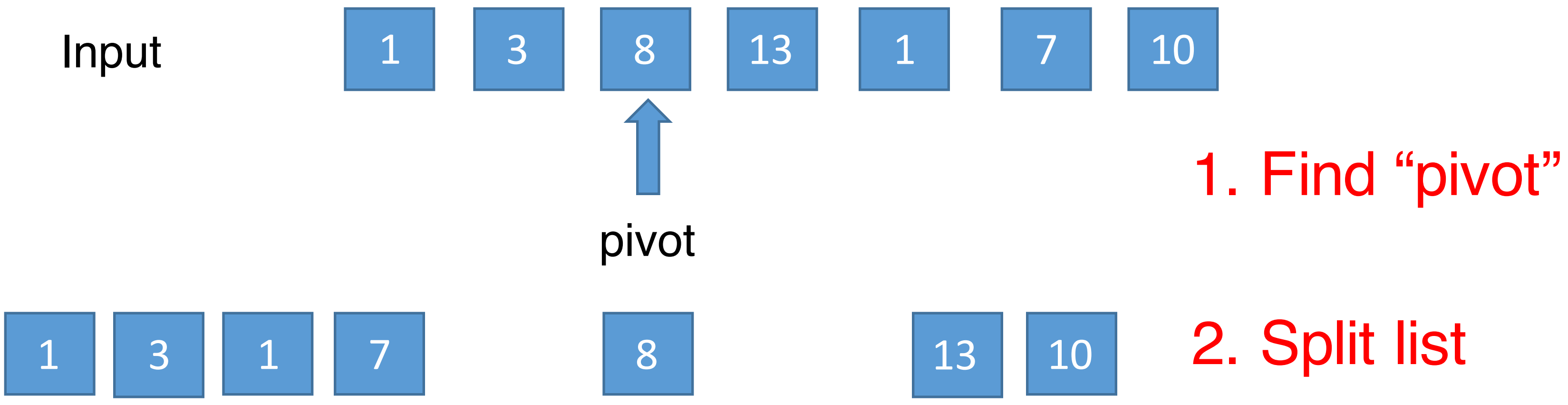
In terms of lambda calculus,

the curried function of $\lambda x_1\ x_2\ \dots\ x_n.e$ is
$\lambda x_1.(\lambda x_{2.}(\dots(\lambda x_n.e)))$

```
(define (curry2 f)
  (lambda (x)
    (lambda (y)
      (f x y))))
```

```
(define (curry3 f)
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (f x y z)))))
```

Quick Sort



Input 1 3 8 13 1 7 10

pivot

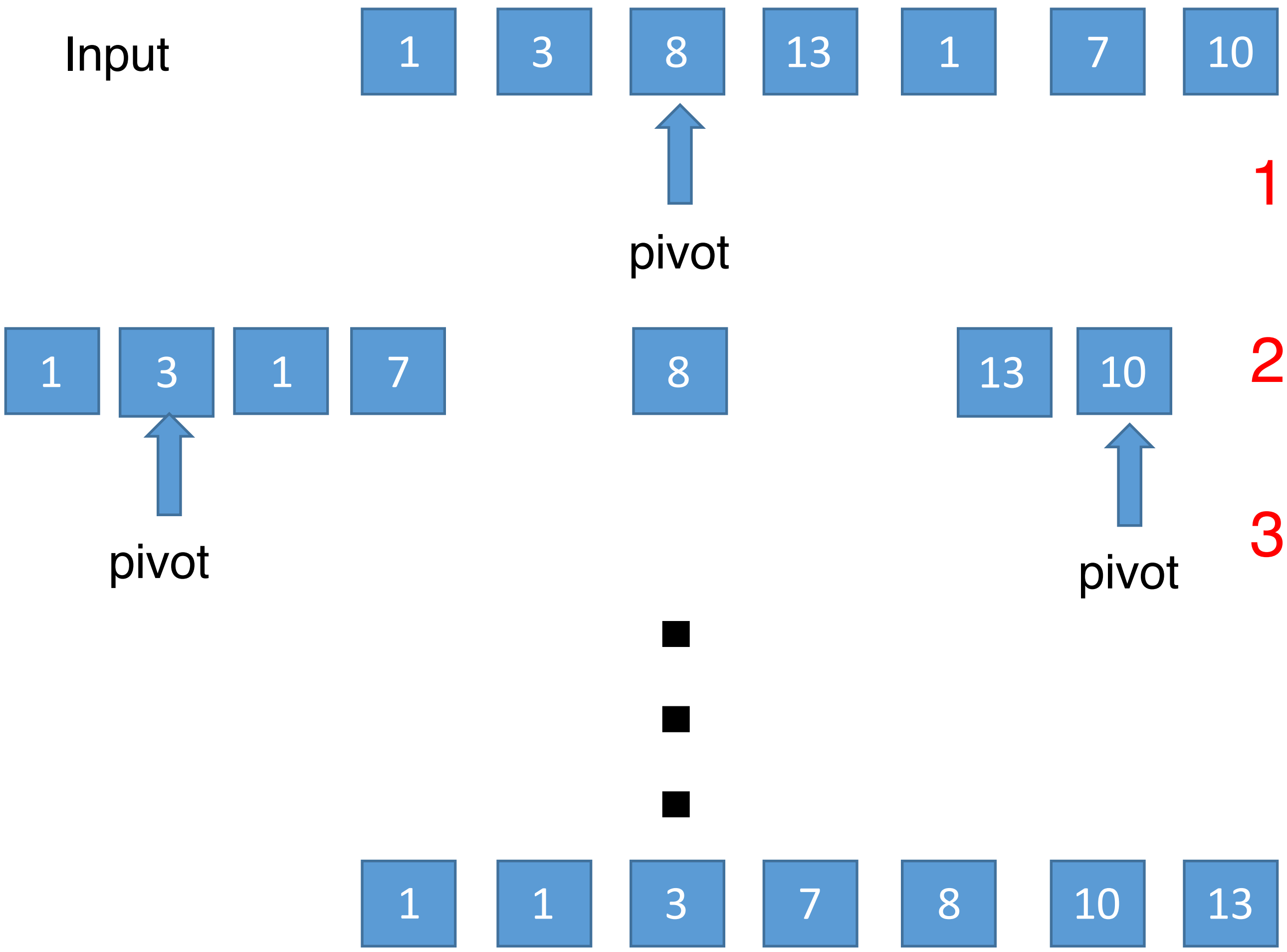
1. Find “pivot”

1 3 1 7 8 13 10

2. Split list

```
(filter ((curry2 >) pivot) lst)

(filter ((curry2 =) pivot) lst)

(filter ((curry2 <) pivot) lst)
```

Quick Sort



1. Find "pivot"

2. Split list

3. Recursion

```
(define (getPivot lst) (list-ref lst (random (length lst))))
(define (quicksort lst)
  (cond ((or (null? lst) (= (length lst) 1)) lst)
    ((let ((pivot (getPivot lst)))
      (append (quicksort (filter ((curry2 >) pivot) lst))
              (filter ((curry2 =) pivot) lst)
              (quicksort (filter ((curry2 <) pivot) lst))
)))))
```

Return decreasing numbers?

Sort a list of strings?

Pass in a comparator (a function defining ordering)

Comparator

A function that defines ordering:

Uncurried: $(\mathrm{Elem}, \mathrm{Elem}) \to \mathrm{Boolean}$

Curried: $\mathrm{Elem} \to \mathrm{Elem} \to \mathrm{Boolean}$

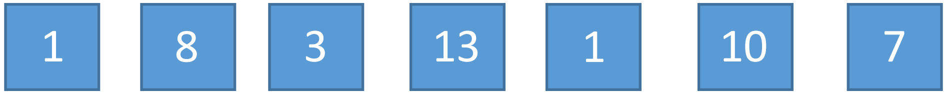
Examples:

<	<=	>
string<?	string<=?	string>=?

comparator

```
(define (quicksort lst lt)
    (cond ((or (null? lst) (= (length lst) 1)) lst)
      ((let ((pivot (getPivot lst))
          (gt (lambda (x y) (not (or (equal? x y) (lt x y))))))
        (append (quicksort (filter ((curry2 gt) pivot) lst) lt)
                (filter ((curry2 equal?) pivot) lst)
                (quicksort (filter ((curry2 lt) pivot) lst) lt)
  )))))
```

Merge Sort



1 8 3 13 1 10 7

1. Split the list

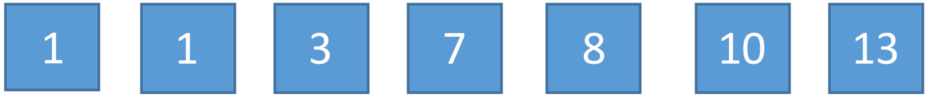
1 8 3 13

1 10 7

2. (Recursively) Sort sublists

1 3 8 13

1 7 10



1 1 3 7 8 10 13

Merge Sort

1	8	3	13	1	10	7

Split the list

```
(define (split lst)
        (fold-left (lambda (prev e)
                   (let ((fst (car prev))
                         (snd (cadr prev)))
                   (if (< (length fst) (/ (length lst) 2))
                      (list (append fst (list e)) snd)
                      (list fst (append snd (list e))))))
        '(() ()) lst))
```

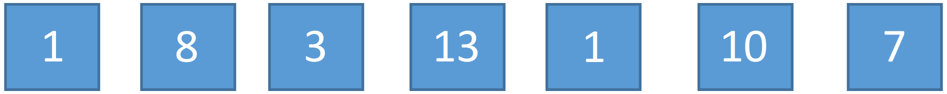
Merge Sort

1 8 3 13 1 10 7

Split the list with even/odd positions

```
(define (split lst)
        (fold-left (lambda (pair e)
                   (let ((fst (car pair))
                        (snd (cadr pair)))
                   (if (<= (length fst) (length snd))
                      (list (append fst (list e)) snd)
                      (list fst (append snd (list e))))))
        '(() ()) lst))
```

Merge Sort


1 8 3 13 1 10 7

Merge lists

```
(define (merge lst1 lst2)
    (cond ((null? lst1) lst2)
          ((null? lst2) lst1)
          (else (if (< (car lst1) (car lst2))
              (cons (car lst1) (merge (cdr lst1) lst2))
              (cons (car lst2) (merge lst1 (cdr lst2)))))))
```

Merge Sort

```
(define (merge-sort lst)
  (cond ((null? lst) lst)
        ((= 1 (length lst)) lst)
        (else (let ((lsts (split lst)))
                 (merge (merge-sort (car lsts))
                        (merge-sort (cadr lsts)))))))
```

Return decreasing numbers?

Sort a list of strings?

Definitions & Scope

A definition (function, value):

- Binds a name to a function, value
- Provides a *scope* (visibility) for that binding

Different rules for *scope* (visibility)

- Static (lexical) scoping
- Dynamic scoping

Scope Rules

x is not bound (not visible)

Scope

```
(let((x 0))
     (+ x x))
```

```
(let ((x 0) (y x))
     (+ x y))
```

Scope of x

```
(let((x 0))
     (let (y (+ x 1)))
          (+ x y)))
```

Scope of y

Scope

```
(define (f x)
  (if (= 0 x) 0
      (+ x (f (- x 1))))))
(f 2)
```

Static scoping in Scheme

Typing

Primitive operations expect data of specific types

- A type is a *collection* of values
- Different types have different representations
- E.g., floats have different format than integers

A language specifies if/when types are checked

- Statically – done at compile time (before execution)
- Dynamically – done at run time (just in time)

Type Checking

```
(define (f x y)
     (/ (+ x y) 3))
(f 2 2)
(f 2.0 2.0)
```

```
(define (f p q)
     (p (q 1 2)))
(f even? +)
(f sqrt  *)
(f 1     +)
```

`1` is not a procedure

```
(define (f x)
  (x + “true”))
(f 2)
```

`true` is not a number

Dynamic tying in Scheme