Higher-Order Functions and Loops

Warm Up: Sorting a List of Numbers

Remember quicksort?

- (quicksort (list 3 9 6 2 1 7))
  - [smaller than pivot] (quicksort (list 2 1))
    - (quicksort (list 1))
      - (list 1)
    - 2
    - (quicksort empty)
  - [pivot] 3
  - [larger than pivot] (quicksort (list 9 6 7))
    - (quicksort (list 6 7))
      - 6
      - (quicksort (list 7))
        - (list 7)
    - 9
    - (quicksort empty)

Answer: (list 1 2 3 6 7 9)

Warm Up: Sorting a List of Numbers

Let's write quicksort.

As usual, start with the template for list[num]

```
;; quicksort : list[num] → list[num]
;; sorts a list of nums into increasing order
(define (quicksort alon)
   (cond [(empty? alon) …]
              [(cons? alon) …
                       (first alon) …
                       (quicksort (rest alon)) … ]))
```

Warm Up: Sorting a List of Numbers

What do the pieces in the cons? case give us?

```
;; quicksort : list[num] → list[num]
;; sorts a list of nums into increasing order
(define (quicksort alon)
  (cond [(empty? alon) …]
        [(cons? alon) …
              (first alon) …
              (quicksort (rest alon)) … ]))
```

a number

sorts the rest of the list into increasing order

Warm Up: Sorting a List of Numbers

So, how do we combine them? We need to insert the first element into the sorted rest of the list …

```
;; quicksort : list[num] → list[num]
;; sorts a list of nums into increasing order
(define (quicksort alon)
  (cond [(empty? alon) …]
        [(cons? alon) …
              (first alon) …
              (quicksort (rest alon)) … ]))
```

a number

sorts the rest of the list into increasing order

But that's insertion sort!

Writing quicksort via templates

We got insertion-sort. What happened?

- With templates, you write programs according to the “natural” recursion
- Insertion-sort is the naturally recursive sort
- Quicksort uses recursion in a different way

Moral: some algorithms need different forms of recursion (“generative recursion” – see HTDP).

Templates aren’t a catch-all for program design
(but they are still very useful for lots of programs)

Quicksort: Take 2

The template is fine until the natural recursion, so we'll take that out and leave the rest intact …

```
;; quicksort : list[num] → list[num]
;; sorts a list of nums into increasing order
(define (quicksort alon)
  (cond [(empty? alon) …]
        [(cons? alon) …
              (first alon) …
              (quicksort (rest alon)) … ]))
```

How did quicksort work? Gather the elts smaller than (first alon); gather those larger; sort; and combine:

Quicksort: Take 2

How did quicksort work? Gather the elts smaller than (first alon); gather those larger; sort; and combine:

```
;; quicksort : list[num] → list[num]
;; sorts a list of nums into increasing order
(define (quicksort alon)
  (cond [(empty? alon) …]
        [(cons? alon) …
         (smaller-than (first alon) (rest alon)) …

         (larger-than (first alon) (rest alon)) … ]))
```

gather the smaller elts

gather the larger elts

[we'll write smaller-than, larger-than later]

Quicksort: Take 2

How did quicksort work? Gather the elts smaller than (first alon); gather those larger; sort; and combine:

```
;; quicksort : list[num] → list[num]
;; sorts a list of nums into increasing order
(define (quicksort alon)
  (cond [(empty? alon) …]
        [(cons? alon) …
           (quicksort (smaller-than (first alon) (rest alon)))

           (quicksort (larger-than (first alon) (rest alon))) ]))
```

sort the smaller elts

sort the larger elts

Quicksort: Take 2

How did quicksort work? Gather the elts smaller than (first alon); gather those larger; sort; and combine:

```
;; quicksort : list[num] → list[num]
;; sorts a list of nums into increasing order
(define (quicksort alon)
  (cond [(empty? alon) …]
        [(cons? alon)
         (append
           (quicksort (smaller-than (first alon) (rest alon)))
           (list (first alon))     [don't forget the pivot!]
           (quicksort (larger-than (first alon) (rest alon))))]))
```

combine the sorted lists into one list

[append (built in) takes any number of lists and "concatenates" them]

Quicksort: Take 2

The main quicksort program

But where are smaller-than and larger-than?

```
;; quicksort : list[num] → list[num]
;; sorts a list of nums into increasing order
(define (quicksort alon)
   (cond [(empty? alon) empty]
         [(cons? alon)
          (append
             (quicksort (smaller-than (first alon) (rest alon)))
             (list (first alon))
             (quicksort (larger-than (first alon) (rest alon))))]))
```

Smaller-than and Larger-than

```
;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
          (cond [(< (first alon) anum)
                  (cons (first alon) (smaller-than anum (rest alon)))]
                 [else (smaller-than anum (rest alon))])]))

;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
          (cond [(> (first alon) anum)
                  (cons (first alon) (larger-than anum (rest alon)))]
                 [else (larger-than anum (rest alon))])]))
```

Smaller-than and Larger-than

```
;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(< (first alon) anum)
                (cons (first alon) (smaller-than anum (rest alon)))]
               [else (smaller-than anum (rest alon))])]))
```

these programs are identical aside from < and >; can't we share the similar code?

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(> (first alon) anum)
                (cons (first alon) (larger-than anum (rest alon)))]
               [else (larger-than anum (rest alon))])]))
```

Normally, we share similar code by creating parameters for the different parts

Sharing Smaller- and Larger-than code

```
;; extract-nums : num list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum (rest alon)))]
               [else (extract-nums anum (rest alon))])]))
```

First, replace the different part with a new name

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(> (first alon) anum)
                (cons (first alon) (larger-than anum (rest alon)))]
               [else (larger-than anum (rest alon))])]))
```

[larger-than here for reference]

Sharing Smaller- and Larger-than code

Next, add the new name as a parameter

```
;; extract-nums : num list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
   (cond [(empty? alon) empty]
            [(cons? alon)
              (cond [(compare (first alon) anum)
                         (cons (first alon) (extract-nums anum compare (rest alon)))]
                        [else (extract-nums anum compare (rest alon))])]))
```

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
   (cond [(empty? alon) empty]
            [(cons? alon)
              (cond [(> (first alon) anum)
                         (cons (first alon) (larger-than anum (rest alon)))]
                        [else (larger-than anum (rest alon))])]))
```

[larger-than here for reference]

Sharing Smaller- and Larger-than code

```
;; extract-nums : num list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

Next, redefine larger-than in terms of extract-nums …

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (extract-nums anum ____ alon))
```

But what can we send as the argument to the compare parameter?

Sharing Smaller- and Larger-than code

```
;; extract-nums : num list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

Next, redefine larger-than in terms of extract-nums …

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (extract-nums anum > alon))
```

We can send the > operator itself!

But what can we send as the argument to the compare parameter?

Sharing Smaller- and Larger-than code

```
;; extract-nums : num list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (extract-nums anum > alon))

;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
  (extract-nums anum < alon))
```

Don't forget smaller-than

Sharing Smaller- and Larger-than code

```
;; extract-nums : num (num num → bool) list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

We need to fix the contract. What's the contract on compare?

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (extract-nums anum > alon))

;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
  (extract-nums anum < alon))
```

Functions are values in Scheme

This means we can pass them as arguments to functions

We can also return them from functions (but hold that thought for now)

Where else can we use extract-nums?

```
;; extract-nums : num (num num → bool) list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

Extract-nums extracts numbers from lists of numbers

What if we wanted to extract all boas that eat pets or mice from a list of boas?

extract-eats-pets-or-mice

```
;; extract-nums : num (num num → bool) list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
   (cond [(empty? alon) empty]
             [(cons? alon)
              (cond [(compare (first alon) anum)
                         (cons (first alon) (extract-nums anum compare (rest alon)))]
                        [else (extract-nums anum compare (rest alon))])]))
```

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
   (cond [(empty? aloboa) empty]
             [(cons? aloboa)
              (cond [(or (symbol=? 'pets (boa-food (first aloboa)))
                                (symbol=? 'mice (boa-food (first aloboa))))
                         (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
                        [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

extract-eats-pets-or-mice/extract-nums

```
;; extract-nums : num (num num → bool) list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

Where do these functions differ?

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (cond [(empty? aloboa) empty]
        [(cons? aloboa)
         (cond [(or (symbol=? ‘pets (boa-food (first aloboa)))
                    (symbol=? ‘mice (boa-food (first aloboa))))
                (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
               [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

extract-eats-pets-or-mice/extract-nums

```
;; extract-nums : num (num num → bool) list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

Let's write one function that captures both

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (cond [(empty? aloboa) empty]
        [(cons? aloboa)
         (cond [(or (symbol=? 'pets (boa-food (first aloboa)))
                    (symbol=? 'mice (boa-food (first aloboa))))
                (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
               [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

How are these two expressions similar?

extract-eats-pets-or-mice/extract-nums

```
;; extract-nums : num (num num → bool) list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

Both do a comparison on the first elt

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (cond [(empty? aloboa) empty]
        [(cons? aloboa)
         (cond [(or (symbol=? 'pets (boa-food (first aloboa)))
                    (symbol=? 'mice (boa-food (first aloboa))))
                (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
               [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

Both expressions return booleans

extract-eats-pets-or-mice/extract-nums

```
;; extract-nums : num (num num → bool) list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compare (first alon) anum)
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

Compares first against one datum

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (cond [(empty? aloboa) empty]
        [(cons? aloboa)
         (cond [(or (symbol=? ‘pets (boa-food (first aloboa)))
                    (symbol=? ‘mice (boa-food (first aloboa))))
                (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
               [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

Compares first against two data

Summary: What's in common?

- Both expressions perform some comparison on the first elt of the list
- Both comparisons return booleans
- But, the expressions use different numbers of additional information in their comparisons

So, to collapse these expressions into a common definition, they need to take the first elt and return a boolean …

extract-eats-pets-or-mice/extract-nums

```
;; extract-nums : num (num num → bool) list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums anum compare alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compares-to-num? (first alon))
                (cons (first alon) (extract-nums anum compare (rest alon)))]
               [else (extract-nums anum compare (rest alon))])]))
```

Rewritten in terms of functions from first → bool

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (cond [(empty? aloboa) empty]
        [(cons? aloboa)
         (cond [(food-is-pets-or-mice? (first aloboa))
                (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
               [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

extract-eats-pets-or-mice/extract-nums

```
;; extract-nums : list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums alon)
   (cond [(empty? alon) empty]
             [(cons? alon)
               (cond [(compares-to-num? (first alon))
                            (cons (first alon) (extract-nums (rest alon)))]
                           [else (extract-nums (rest alon))])]))
```

Remove compare and anum parameters since extract-nums no longer uses them

---

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
   (cond [(empty? aloboa) empty]
             [(cons? aloboa)
               (cond [(food-is-pets-or-mice? (first aloboa))
                            (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
                           [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

extract-eats-pets-or-mice/extract-nums

```
;; extract-nums : list[num] → list[num]
;; returns elts in input list that compare to the given num
(define (extract-nums alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(compares-to-num? (first alon))
                (cons (first alon) (extract-nums (rest alon)))]
               [else (extract-nums (rest alon))])]))
```

Now, these two functions look identical minus the name of the comparison function …

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (cond [(empty? aloboa) empty]
        [(cons? aloboa)
         (cond [(food-is-pets-or-mice? (first aloboa))
                (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
               [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

extract-elts

```
;; extract-elts : list[num] → list[num]
;; returns elts in input list that satisfy keep? predicate
(define (extract-elts keep? alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(keep? (first alon))
                (cons (first alon) (extract-elts keep? (rest alon)))]
               [else (extract-elts keep? (rest alon))])]))
```

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (cond [(empty? aloboa) empty]
        [(cons? aloboa)
         (cond [(food-is-pets-or-mice? (first aloboa))
                (cons (first aloboa) (extract-eats-pets-or-mice (rest aloboa)))]
               [else (extract-eats-pets-or-mice (rest aloboa))])]))
```

Make the name of the comparison function a parameter. We use keep? Since the comparison determines whether we keep an elt in the output

extract-elts and extract-eats

```
;; extract-elts : list[num] → list[num]
;; returns elts in input list that satisfy keep? predicate
(define (extract-elts keep? alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(keep? (first alon))
                (cons (first alon) (extract-elts keep? (rest alon)))]
               [else (extract-elts keep? (rest alon))])]))
```

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (extract-elts food-is-pets-or-mice? aloboa))

;; food-is-pets-or-mice? : boa → boolean
;; determines whether boa's food is 'pets or 'mice
(define (food-is-pets-or-mice? aboa)
  (or (symbol=? (boa-food aboa) 'pets)
      (symbol=? (boa-food aboa) 'mice)))
```

Redefine extract-eats in terms of extract-elts

extract-elts and extract-eats

```
;; extract-elts : list[num] → list[num]
;; returns elts in input list that satisfy keep? predicate
(define (extract-elts keep? alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(keep? (first alon))
                (cons (first alon) (extract-elts keep? (rest alon)))]
               [else (extract-elts keep? (rest alon))])]))
```

Notice the contracts don't match up though!

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (extract-elts food-is-pets-or-mice? aloboa))

;; food-is-pets-or-mice? : boa → boolean
;; determines whether boa's food is 'pets or 'mice
(define (food-is-pets-or-mice? aboa)
  (or (symbol=? (boa-food aboa) 'pets)
      (symbol=? (boa-food aboa) 'mice)))
```

extract-elts and extract-eats

```
;; extract-elts : list[α] → list[num]
;; returns elts in input list that satisfy keep? predicate
(define (extract-elts keep? alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(keep? (first alon))
                (cons (first alon) (extract-elts keep? (rest alon)))]
               [else (extract-elts keep? (rest alon))])]))
```

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (extract-elts food-is-pets-or-mice? aloboa))

;; food-is-pets-or-mice? : boa → boolean
;; determines whether boa's food is 'pets or 'mice
(define (food-is-pets-or-mice? aboa)
  (or (symbol=? (boa-food aboa) 'pets)
      (symbol=? (boa-food aboa) 'mice)))
```

Nothing in the defn of extract-elts requires numbers, so we can relax the contract to allow input lists of any type. α is just a variable over types.

extract-elts and extract-eats

```
;; extract-elts : list[α] → list[α]
;; returns elts in input list that satisfy keep? predicate
(define (extract-elts keep? alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(keep? (first alon))
                (cons (first alon) (extract-elts keep? (rest alon)))]
               [else (extract-elts keep? (rest alon))])]))
```

Since the output list contains elements of the input list, the type of the output list should also refer to α …

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (extract-elts food-is-pets-or-mice? aloboa))

;; food-is-pets-or-mice? : boa → boolean
;; determines whether boa's food is 'pets or 'mice
(define (food-is-pets-or-mice? aboa)
  (or (symbol=? (boa-food aboa) 'pets)
      (symbol=? (boa-food aboa) 'mice)))
```

extract-elts and extract-eats

```
;; extract-elts : list[α] → list[α]
;; returns elts in input list that satisfy keep? predicate
(define (extract-elts keep? alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(keep? (first alon))
                (cons (first alon) (extract-elts keep? (rest alon)))]
               [else (extract-elts keep? (rest alon))])]))
```

We also never added keep? to the contract.

What is keep?'s type?

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (extract-elts food-is-pets-or-mice? aloboa))

;; food-is-pets-or-mice? : boa → boolean
;; determines whether boa's food is 'pets or 'mice
(define (food-is-pets-or-mice? aboa)
  (or (symbol=? (boa-food aboa) 'pets)
      (symbol=? (boa-food aboa) 'mice)))
```

extract-elts and extract-eats

```
;; extract-elts : (α → bool) list[α] → list[α]
;; returns elts in input list that satisfy keep? predicate
(define (extract-elts keep? alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (cond [(keep? (first alon))
                (cons (first alon) (extract-elts keep? (rest alon)))]
               [else (extract-elts keep? (rest alon))])]))
```

We also never added keep? to the contract.

What is keep?'s type?

```
;; extract-eats-pets-or-mice : list[boa] → list[boa]
;; returns boas in input list that eat pets or mice
(define (extract-eats-pets-or-mice aloboa)
  (extract-elts food-is-pets-or-mice? aloboa))

;; food-is-pets-or-mice? : boa → boolean
;; determines whether boa's food is 'pets or 'mice
(define (food-is-pets-or-mice? aboa)
  (or (symbol=? (boa-food aboa) 'pets)
      (symbol=? (boa-food aboa) 'mice)))
```

keep? takes an elt of the list and returns a boolean.

Filter

extract-elts is built-in. It's called filter

```
;; filter : (α → bool) list[α] → list[α]
;; returns list of elts in input list that satisfy keep? predicate
(define (filter keep? alst)
   (cond [(empty? alst) empty]
             [(cons? alst)
               (cond [(keep? (first alst))
                           (cons (first alst) (filter keep? (rest alst)))]
                          [else (filter keep? (rest alst))])]))
```

Use filter whenever you want to extract elts from a list according to some predicate

Back to smaller-than and larger-than

Rewrite these in terms of filter …

```
;; filter : (α → bool) list[α] → list[α]
```

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (extract-nums anum > alon))

;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
  (extract-nums anum < alon))
```

Must replace the calls to extract-nums with calls to filter

Back to smaller-than and larger-than

Rewrite these in terms of filter …

;; filter : (α → bool) list[α] → list[α]

;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
   (filter ________ alon))
  ;; was (extract-nums anum > alon))

What do we pass as keep?

;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
   (extract-nums anum < alon))

Need a function that consumes a num and returns a bool; function must compare input to anum …

Back to smaller-than and larger-than

Rewrite these in terms of filter …

```
;; filter : (α → bool) list[α] → list[α]
```

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
   (filter (make-function (elt) (> elt anum))
           alon))
  ;; was (extract-nums anum > alon))

;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
   (extract-nums anum < alon))
```

We'd like something like make-function that takes a list of parameters and the body of the function

Need a function that consumes a num and returns a bool; function must compare input to anum …

Back to smaller-than and larger-than

Rewrite these in terms of filter …

;; filter : (α → bool) list[α] → list[α]

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (filter (lambda (elt) (> elt anum))
          alon))
 ;; was (extract-nums anum > alon))

;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
  (extract-nums anum < alon))
```

make-function exists in Scheme …

It's called *lambda*

Back to smaller-than and larger-than

Rewrite these in terms of filter …

;; filter : (α → bool) list[α] → list[α]

```
;; larger-than : num list[num] → list[num]
;; returns elts in input list that are larger than given num
(define (larger-than anum alon)
  (filter (lambda (elt) (> elt anum))
          alon))
```

We can rewrite smaller-than in the same way

```
;; smaller-than : num list[num] → list[num]
;; returns elts in input list that are smaller than given num
(define (smaller-than anum alon)
  (filter (lambda (elt) (< elt anum))
          alon))
```

Summary: What have we seen?

- Functions are values and can be passed as arguments to other functions
  - This lets us share code between similar functions
- Scheme provides lambda to make new functions
- We can pass functions created with define or functions created with lambda as arguments

Actually, (define (square n) (* n n)) is a shorthand for
(define square (lambda (n) (* n n)))

Summary: What have we seen?

- We’ve also seen filter, which takes a function (predicate) and a list and returns a list of elements in the list for which the function returns true.
- Filter provides a nice, compact way of writing certain Scheme functions

Using Filter

```
;; filter : (α → bool) list[α] → list[α]
;; returns list of elts in input list that satisfy keep? predicate
(define (filter keep? alst)
   (cond [(empty? alst) empty]
             [(cons? alst)
              (cond [(keep? (first alst))
                         (cons (first alst) (filter keep? (rest alst)))]
                        [else (filter keep? (rest alst))])]))
```

Let's use filter to get the list of all foods that a list of boas will eat

```
Example: (all-foods (list (make-boa ‘Slinky 10 ‘pets)
                          (make-boa ‘Curly 55 ‘rice)
                          (make-boa ‘Slim 15 ‘lettuce)))
         = (list ‘pets ‘rice ‘lettuce)
```

Using Filter

```
;; filter : (α → bool) list[α] → list[α]
;; returns list of elts in input list that satisfy keep? predicate
(define (filter keep? alst)
   (cond [(empty? alst) empty]
             [(cons? alst)
               (cond [(keep? (first alst))
                          (cons (first alst) (filter keep? (rest alst)))]
                         [else (filter keep? (rest alst))])]))
```

```
;; all-foods : list[boa] → list[symbol]
;; given a list of boas, extracts a list of the foods that the boas eat
(define (all-foods aloboa)
    (filter (lambda (aboa) …) aloboa))
```

[What goes in the body of the lambda?]

Using Filter

```
;; filter : (α → bool) list[α] → list[α]
;; returns list of elts in input list that satisfy keep? predicate
(define (filter keep? alst)
   (cond [(empty? alst) empty]
             [(cons? alst)
               (cond [(keep? (first alst))
                          (cons (first alst) (filter keep? (rest alst)))]
                         [else (filter keep? (rest alst))])]))
```

```
;; all-foods : list[boa] → list[symbol]
;; given a list of boas, extracts a list of the foods that the boas eat
(define (all-foods aloboa)
    (filter (lambda (aboa) (boa-food aboa)) aloboa))
```

How about we simply extract the boa's food?

Using Filter

```
;; filter : (α → bool) list[α] → list[α]
;; returns list of elts in input list that satisfy keep? predicate
(define (filter keep? alst)
  (cond [(empty? alst) empty]
        [(cons? alst)
         (cond [(keep? (first alst))
                (cons (first alst) (filter keep? (rest alst)))]
               [else (filter keep? (rest alst))])]))
```

```
;; all-foods : list[boa] → list[symbol]
;; given a list of boas, extracts a list of the foods that the boas eat
(define (all-foods aloboa)
  (filter (lambda (aboa) (boa-food aboa)) aloboa))
```

How about we simply extract the boa's food?

Look at the contract – does boa-food return a boolean?

No, it returns a symbol …

Also, filter would return list[boa], not list[symbol] …

Using Filter

```
;; filter : (α → bool) list[α] → list[α]
;; returns list of elts in input list that satisfy keep? predicate
(define (filter keep? alst)
  (cond [(empty? alst) empty]
        [(cons? alst)
         (cond [(keep? (first alst))
                (cons (first alst) (filter keep? (rest alst)))]
               [else (filter keep? (rest alst))])]))
```

Filter returns a list of the same type as the input list, and is designed to leave some elements out.

all-foods must return a list of a different type, but with information gathered from *every* element of the input list

We need another function that takes a function and a list (like filter does), but with slightly different behavior

Map: Transforms a list

```
;; map : (α → β) list[α] → list[β]
;; returns list of results from applying function to every elts in input list
(define (map f alst)
  (cond [(empty? alst) empty]
        [(cons? alst) (cons (f (first alst)) (map f (rest alst)))]))
```

Filter returns a list of the same type as the input list, and is designed to leave some elements out.

all-foods must return a list of a different type, but with information gathered from *every* element of the input list

We need another function (map) that takes a function and a list (like filter does), but with slightly different behavior

Implementing all-foods with map

```
;; map : (α → β) list[α] → list[β]
;; returns list of results from applying function to every elts in input list
(define (map f alst)
  (cond [(empty? alst) empty]
        [(cons? alst) (cons (f (first alst)) (map f (rest alst)))]))
```

```
;; all-foods : list[boa] → list[symbol]
;; given a list of boas, extracts a list of the foods that the boas eat
(define (all-foods aloboa)
  (map ____________ aloboa))
```

What function do we want to apply to each boa?

Implementing all-foods with map

```
;; map : (α → β) list[α] → list[β]
;; returns list of results from applying function to every elts in input list
(define (map f alst)
  (cond [(empty? alst) empty]
        [(cons? alst) (cons (f (first alst)) (map f (rest alst)))]))
```

```
;; all-foods : list[boa] → list[symbol]
;; given a list of boas, extracts a list of the foods that the boas eat
(define (all-foods aloboa)
  (map (lambda (aboa) (boa-food aboa)) aloboa))
```

What function do we want to apply to each boa?

boa-food

Implementing all-foods with map

```
;; map : (α → β) list[α] → list[β]
;; returns list of results from applying function to every elts in input list
(define (map f alst)
  (cond [(empty? alst) empty]
        [(cons? alst) (cons (f (first alst)) (map f (rest alst)))]))
```

```
;; all-foods : list[boa] → list[symbol]
;; given a list of boas, extracts a list of the foods that the boas eat
(define (all-foods aloboa)
  (map boa-food aloboa))
```

Actually, we could write this more concisely
(since boa-food is already a function from boa → symbol)

Using map and filter together

Given a zoo (a list of boas and armadillos), how can we get the list of all foods eaten by the boas (ignoring the armadillos)?

```
;; all-boa-foods : list[animal] → list[symbol]
;; given a list of animals, extracts a list of the foods that the
;;     boas eat
(define (all-boa-foods aloboa)
  (map boa-food (filter boa? aloboa)))
```

Summary

- map and filter are Scheme's looping constructs
- Each loops over the elements of a list
  - filter extracts elements according to a predicate
  - map applies a function to every element
- Their names are descriptive, in that they tell you what kind of operation the loop performs (in contrast to **while** versus **repeat** versus **for** loops in other languages)
- From now on, use map and filter in your programs