# CS115 - Module 7 - Processing two lists or numbers

Cameron Morland

Fall 2017

Reminder: if you have not already, ensure you:

- Read *How to Design Programs*, Section 17.

# Processing two lists simultaneously

Sometimes we have data in more than one separate list, and need to do computation on the lists together.
We identify three important cases:

A list "going along for the ride" E.g. appending two lists:

```
(my-append (list 1 2 3) (list 4 5 6)) ⇒ (list 1 2 3 4 5 6)
```

Processing "in lockstep" E.g. adding items in one list to corresponding items in another:

```
(add-pairs (list 1 2 3) (list 5 8 6)) ⇒ (list 6 10 9)
```

Processing at different rates E.g. merging two sorted lists:

```
(merge (list 2 3 7) (list 4 6 8 9)) ⇒ (list 2 3 4 6 7 8 9)
```

# Adding to a list

Inserting an item at the front of a list is easy: `(cons 7 (list 5 3 2))` ⇒ `(list 7 5 3 2)`
Appending an item at the back requires a little recursion:

```
;; (add-end n L) add n at the end of L.
;; add-end: Num (listof Any) -> (listof Any)
;; Example:
(check-expect (add-end 7 (list 2 3 5)) (list 2 3 5 7))

(define (add-end n L) (cond [(empty? L) (cons n empty)]
                            [else (cons (first L) (add-end n (rest L)))]))
```

How much harder would it be to append a list instead of just a number?

**Exercise:** Complete `append-lists`

```
;; (append-lists L1 L2) form a list of the items in L1 then L2, in order.
;; append-lists: (listof Any) (listof Any) -> (listof Any)
;; Example:
(check-expect (append-lists (list 3 7 4) (list 3 6 8)) (list 3 7 4 3 6 8))
```

## Template for a list "going along for the ride"

We do not need to recurse through `L2` in order to append it to `L1`. `L2` is present in the recursion, and is passed to the next recursive call.

We use **`first`** and **`rest`** on `L1`, just like in single-list recursion.

The template looks like this:

```
(define (my-alongforride-fun L1 L2)
  (cond
    [(empty? L1) ... ]
    [else (... (first L1) ...
               ... (my-alongforride-fun (rest L1) L2) ...)]))
```

# Another list "going along for the ride"

We can instead recurse on a number, with an unchanged list:

```
;; (copy-list L n) return a list with n copies of L.
;; copy-list: (listof Any) Nat
;; Example:
(check-expect (copy-list (list 42 6 7) 3)
              (list (list 42 6 7) (list 42 6 7) (list 42 6 7)))
```

**Exercise:** Complete `copy-list`.

# Dot-product

We may process two lists of the same length, at the same time.

The dot product of two vectors is the sum of the products of the corresponding elements of the vectors. (This works for vectors of any dimension.)
E.g. if $\vec{u} = [2, 3, 5]$ and $\vec{v} = [7, 11, 13]$, then $\vec{u} \cdot \vec{v} = 2 \cdot 7 + 3 \cdot 11 + 5 \cdot 13 = 112$.

**Exercise:** Complete `dot-product`.

```
;; A Vector is a (listof Num).

;; (dot-produce u v) return the dot product of u and v.
;; dot-product: Vector Vector -> Num
;; Requires: u and v have the same length.
;; Example:
(check-expect (dot-product (list 2 3 5) (list 7 11 13)) 112)
```

## Lockstep template

Here we are consuming the two lists at the same rate, and they are of the same length. When one becomes empty, the other does too.

```
(define (lockstep-fun L1 L2)
  (cond
   [(empty? L1) ... ] ; We could check (and (empty L1) (empty L2).

   [else (... (first L1) ... (first L2) ... ; We use both firsts.

              ... (lockstep-fun (rest L1) (rest L2)) ... )]))
              ; We make a recursive call on both rests.
```

**Exercise:** Write a function `vector-add` that adds two vectors.
For example: `(vector-add (list 3 5) (list 7 11)) => (list 10 16)`

# Merging two sorted lists

Suppose I have two lists, each sorted, and I wish to create a sorted list that contains the items from both lists.

`(merge (list 2 3 7) (list 4 6 8 9))` $\Rightarrow$ `(list 2 3 4 6 7 8 9)`

Idea: look at the **`first`** item in both lists. Take the smaller one; use **`rest`** to remove it from one of the lists, then finish the problem using recursion.

**Exercise:** Complete **`merge`**.

```
;; (merge L1 L2) return the list of all items in L1 and L2, in order.
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: L1 is sorted; L2 is sorted.
;; Example:
(check-expect (merge (list 2 3 7) (list 4 6 8 9)) (list 2 3 4 6 7 8 9))
```

# Generic two-list template

More generally, we may need to consider if both lists are empty, if just one is empty, or if both are non-empty.

If `L` is a list, `(cons? L)` gives the same answer as `(not (empty? L))`.

```
(define (my-two-list-fn L1 L2)
  (cond
    [(and (empty? L1)
          (empty? L2))
     ... ]
    [(and (empty? L1)
          (cons? L2))
     (... (first L2) ... (rest L2) ...)]
    [(and (cons? L1)
          (empty? L2))
     (... (first L1) ... (rest L1) ...)]
    [(and (cons? L1)
          (cons? L2))
     (... ??? ...) ]))
```

```
(define (my-two-list-fn L1 L2)
  (cond
    [(and (empty? L1)
          (empty? L2))
     ... ]
    [(and (empty? L1)
          (not (empty? L2)))
     (... (first L2) ... (rest L2) ...)]
    [(and (not (empty? L1))
          (empty? L2))
     (... (first L1) ... (rest L1) ...)]
    [(and (not (empty? L1))
          (not (empty? L2)))
     (... ??? ...) ]))
```

# Some examples using prime factor decomposition (pfd)

```
;; A PFD, or prime factor decomposition, is a (listof Nat)
;; Requires:
;;   the elements are in ascending order
;;   the elements are prime numbers.

;; (factorize n) return the prime factor decomposition of n.
;; factorize: Nat -> PFD
;; Examples:
(check-expect (factorize 1)  empty)
(check-expect (factorize 17) (list 17))
(check-expect (factorize 24) (list 2 2 2 3))
(check-expect (factorize 42) (list 2 3 7))
```

**Exercise:** Complete `factorize`. It may be helpful to consider the `count-up` template for recursion on a `Nat`.

# Greatest common divisor (gcd) using pfd

Given the prime factor decomposition of two numbers, it is relatively easy to compute the gcd. This can be solved using the generic two-list template.

```
;; (pfd-gcd p1 p2) return the gcd of p1 and p2.
;; pfd-gcd: PFD PFD -> PFD
;; Examples:
(check-expect (pfd-gcd (list 2 2 3) (list 2 3 3 5)) (list 2 3))
(check-expect (pfd-gcd (list 2 3 5) (list 3 3 7)) (list 3))
(check-expect (pfd-gcd (list 5 7) (list 3 11)) empty)
(check-expect (pfd-gcd (list 5 7) empty) empty)
```

**Exercise:** Complete `pfd-gcd`.

**Exercise:** Create `pfd-lcm`, which computes the Least Common Multiple of two `PFD`s.

## Midpoints of pairs of `Posn`

Suppose we have two lists of `Posn` and we wish to find the midpoint between all the pairs.

```
;; (posn-midpoints L1 L2) return pairwise midpoints of L1 and L2.
;; posn-midpoints: (listof Posn) (listof Posn) -> (listof Posn)
;; Example:
(check-expect (posn-midpoints (list (make-posn 1 2) (make-posn 6 8))
                              (list (make-posn 5 2) (make-posn 10 2)))
              (list (make-posn 3 2) (make-posn 8 5)))
```

We will consume one item from each list to make one new item. Use the lockstep template!

We need to be able to create the midpoint of two `Posn`. Good place for a helper function:

```
;; (mid-posn p1 p2) return the midpoint between P1 and P2.
;; mid-posn: Posn Posn -> Posn
(define (mid-posn p1 p2)
  (make-posn (/ (+ (posn-x p1) (posn-x p2)) 2)
             (/ (+ (posn-y p1) (posn-y p2)) 2)))
```

**Exercise:** Complete `posn-midpoints`.

## From `pfd-gcd` to `pfd-lcm`

```
;; (pfd-gcd p1 p2) return the gcd of p1 and p2.
;; pfd-gcd: PFD PFD -> PFD
;; Examples:
(check-expect (pfd-gcd (list 2 2 3) (list 2 3 3 5)) (list 2 3))

(define (pfd-gcd p1 p2)
  (cond [(or (empty? p1) (empty? p2)) empty]
        [(= (first p1) (first p2))
         (cons (first p1) (pfd-gcd (rest p1) (rest p2)))]
        [(< (first p1) (first p2)) (pfd-gcd (rest p1) p2)]
        [(> (first p1) (first p2)) (pfd-gcd p1 (rest p2))]))
```

**Exercise:** Complete `pfd-lcm`.

```
;; (pfd-lcm L1 L2) return the lcm of p1 and p2.
;; pfd-lcm: PFD PFD -> PFD
;; Example:
(check-expect (pfd-lcm (list 2) (list 2)) (list 2))
(check-expect (pfd-lcm (list 2 2 3) (list 2 3 3 5)) (list 2 2 3 3 5))
```

# List equality

How can we tell if two lists are the same?
The built-in function `equal?` will do it, but let's write our own.
Things to consider:

- Base case: if one list is empty, and the other isn't, they're not equal.
- If the first items aren't equal, the lists aren't equal.
- The empty list is equal to itself.

**Exercise:** Complete **`list`**`=?`

```
;; (list=? a b) return true iff a and b are equal.
;; list=?: (listof Any) (listof Any) -> Bool
;; Examples:
(check-expect (list=? (list 6 7 42) (list 6 7 42)) true)
```

**Exercise:** For added enjoyment (!), rewrite **`list`**`=?` without using **`cond`**.

## Module summary

Be comfortable writing functions that recurse on two lists (or other types). We discussed three types of such recursive functions:

1. consuming one list, with the other "going along for the ride"
2. consuming two lists of equal length, at the same rate.
3. consuming two lists, possibly at different rates (the more general case)

Be able to choose which type to use to solve a given problem!

Before we begin the next module, please

- Read *How to Design Programs*, Section 14.