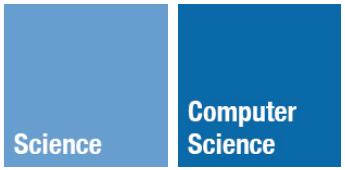
Inlab 1: List Processing (I)



CS 116: Object Oriented Programming II

Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Goals

- Use `cons`, `first`, `rest`, `empty` to create lists and access list elements
- Have a good feel for the "shape" of list processing functions
- Be able to write functions that iterate over and process lists

Classes of list processing functions

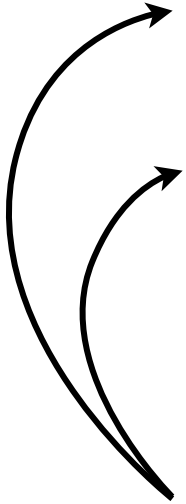
1. Functions that consume one or more lists and *reduce* them to discrete values (e.g., numbers or Booleans)
2. Functions that consume one or more lists and return lists

How to design these functions?

Lists are of *arbitrary size*

- futile to write a function to process *all* the elements of a list

So, write a function that:



1. *processes* a single element
2. handles the rest with *another function call* (often to itself)

also need to *combine* the results of these operations

But can't indiscriminately process the "rest" of the list!

Will either:

1. run out of list elements (error)
2. recurse infinitely

Must test for and handle a *base case*

- return immediately (no recursion)
- may have *1 or more* base cases!
  (with different return values)

```
;; typical list processing fn shape
(define (lst-fn lst)
  (if (empty? lst)
      ...
      (... (first lst) ...
           ... (lst-fn (rest lst)) ...)))
```

To be decided:

- base case test(s)/return value(s)?
- how to process a single element?
- how to combine results?

Design `sum`

- input: list of numbers
- output: (reduced) sum of numbers
- e.g., `(sum (list 1 2 3 4))` ⇒ `10`
  `(sum empty)` ⇒ `0`

`sum` implementation checklist:

- base case test(s)/return value(s)?
- how to process a single element?
- how to combine results?

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

Design `product`

- input: list of numbers
- output: product of numbers
- e.g., `(product (list 1 2 3 4))` ⇒ `24`
  `(product empty)` ⇒ `1`

Design `count`

- input: list
- output: count of items in list
- e.g., `(count (list 1 2 3))` ⇒ `3`
  `(count (list 'red 10))` ⇒ `2`
  `(count empty)` ⇒ `0`

Design `avg`

- input: non-empty list of numbers
- output: average of numbers in list
- e.g., `(avg (list 70 70 100))` ⇒ `80`
  `(avg (list 100))` ⇒ `100`

Design `concat`

- input: list of strings
- output: all strings, appended together
- e.g., `(concat (list "a" "b" "c"))` ⇒ `"abc"`
  `(concat empty)` ⇒ `""`

Design `all-pos?`

- input: list of numbers
- output: true iff *all* numbers are positive
- e.g.,
```
(all-pos? (list 1 2 3)) ⇒ true
(all-pos? (list -10 5)) ⇒ false
(all-pos? empty) ⇒ true
```

Design `some-pos?`

- input: list of numbers
- output: true iff *some* numbers are positive
- e.g.,
```
(some-pos? (list 1 2 3)) ⇒ true
(some-pos? (list -10 5)) ⇒ true
(some-pos? (list -1 -2)) ⇒ false
(some-pos? empty) ⇒ true
```

Design `contains?`

- input: number, list of numbers
- output: true iff the number is in the list
- e.g., `(contains? 2 (list 1 2 3))` ⇒ `true`
  `(contains? 5 (list 1 2 3))` ⇒ `false`
  `(contains? 100 empty)` ⇒ `false`

Design `count-of`

- input: number, list of numbers
- output: count of given number in list
- e.g., `(count-of 1 (list 1 2 1))` ⇒ `2`
  `(count-of 1 (list -10 5))` ⇒ `0`
  `(count-of 1 empty)` ⇒ `0`

Design `max`

- input: non-empty list of numbers
- output: maximum number
- e.g.,
```
(some-pos? (list 1 2 3)) ⇒ true
(some-pos? (list -10 5)) ⇒ true
(some-pos? (list -1 -2)) ⇒ false
(some-pos? empty) ⇒ true
```

Design `ascending?`

- input: list of numbers
- output: true iff the numbers in the list are in ascending order
- e.g., `(ascending? (list 1 2 3))` ⇒ `true`
  `(ascending? (list 1 3 2))` ⇒ `false`
  `(ascending? empty)` ⇒ `true`

Classes of list processing functions

1. Functions that consume one or more lists and *reduce* them to discrete values (e.g., numbers or Booleans)

2. Functions that consume one or more lists and return lists

Functions of the second class *produce* lists

i.e., they must use `cons`

Same questions:

- base case test?
- how to process first item?
- how to combine results?
  - typically, cons belongs as part of the result "combining" mechanism

Design `negate`

- input: list of numbers
- output: negated list of numbers
- e.g., `(negate (list 1 2 3))` ⇒ `(list -1 -2 -3)`
  `(negate (list -5 5))` ⇒ `(list 5 -5)`
  `(negate empty)` ⇒ `empty`

Design `lengths`

- input: list of strings
- output: list of string lengths
- e.g., `(lengths (list "abc" "d" ""))`
  ⇒ `(list 3 1 0)`
  `(lengths empty)` ⇒ `empty`

Design `passing`

- input: list of numbers
- output: list of numbers ≥ 60
- e.g., `(passing (list 55 75 95 48))`
  ⇒ `(list 75 95)`
  `(passing (list 30 40))` ⇒ `empty`
  `(passing empty)` ⇒ `empty`

Design `take`

- input: number (n), list of numbers
- output: a list of at most n numbers (from the start of the list)
- e.g., `(take 3 (list 10 20 30 40 50))`
  ⇒ `(list 10 20 30)`
  `(take 2 (list 90))` ⇒ `(list 90 80)`
  `(take 100 empty)` ⇒ `empty`