Announcements

• Assignment 4 is due on **Tuesday, October 8, at 9:00 PM.**

• The times and location of office hours are posted on the “Office and Consulting Hours” pages of the course website. Please email us at “cs135@uwaterloo.ca” to set up an appointment outside of these hours

Goals of this tutorial

You should be able to...

• understand and write data definitions for lists

• understand and use the template for processing lists to write recursive functions consuming this type of data.
Review: List data definition

;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))

From the data definition, a list of values of any type is either empty or it consists of a first value followed by a list of values (the rest of the list). This is a recursive definition. It contains a base case, and a recursive (self-referential) case. Recursive types should be processed with recursive functions.

Review: Basic list constructs

- empty: A value representing a list with 0 items.
- cons: Consumes an item and a list and produces a new, longer list.
- first: Consumes a nonempty list and produces the first item.
- rest: Consumes a nonempty list and produces the same list without the first item.
- empty?: Consumes a value and produces true if it is empty and false otherwise.
- cons?: Consumes a value and produces true if it is a cons value and false otherwise.
- list?: Consumes a value and produces true if it is a list value and false otherwise.

Review: Substitution rules

If a is a value and b is a list value:

(first (cons a b)) ⇒ a
(rest (cons a b)) ⇒ b
(empty? empty) ⇒ true
(empty? a) ⇒ false, where a is any value other than empty
(cons? (cons a b)) ⇒ true
(cons? a) ⇒ false, where a is any value not created using cons
(cons? empty) ⇒ false
(list? empty) ⇒ true
Clicker Question 1: List Templates
Which of the following goes in the blank for the listof-X-template?

;; listof-X-template: (listof X) → Any
(define (listof-X-template loX)
  (cond
   [(empty? loX) . . . ]
   [else . . . ]
  ))
A (. . . (first loX) . . . (rest loX) . . . )
B (. . . (first loX) . . . (listof-X-template loX) . . . )
C (. . . (rest loX) . . . (listof-X-template (first loX)) . . . )
D (. . . (first loX) . . . (listof-X-template (rest loX)) . . . )
E (. . . (first loX) . . . (listof-X-template (rest loX)) . . . loX . . . )

Problem 1 - sum-num
As a warmup, based on the previous template, write a function sum-num that consumes a list of numbers and produces the sum of those numbers.

(sum-num (cons 7 (cons 8 (cons 9 empty)))) ⇒ 24
(sum-num (cons 8 (cons 0 (cons 0 (cons 8 (cons −5 empty)))))) ⇒ 11

Problem 1 - sum-num - Design recipe
;; (sum-num lon) Produces the sum of all values in the list lon.
;; sum-num: (listof Num) → Num
;; Examples:
(check-expect (sum-num (cons 5 (cons 2 empty))) 7)
(check-expect (sum-num (cons .2 (cons .14 empty))) .34)

(define (sum-num lon) . . . )
;; Tests:
(check-expect (sum-num (cons 21 (cons −4 (cons .5 empty)))) 17.5)
Problem 2 - Factorial

Mathematically, the formulas for factorial are as follows.

\[ n! = n(n-1)(n-2)(n-3) \ldots (2)(1) \].

Write a function, `factorial`, that consumes a positive natural number and determines its factorial.

```
(factorial 3) => 6
(factorial 6) => 720
```

Problem 2 - Factorial - Design Recipe

```
;;; (factorial n) Consumes a number n and produces its factorial.
;;; factorial: Nat -> Nat
;;; require: n should be a positive natural number.
;;; Examples:
(check-expect (factorial 5) 120)

(define (factorial n) ... )

;;; Tests:
(check-expect (factorial 1) 1)
```

Problem 3 - strings-equal?

Based on the previous template for list of X, write a function `strings-equal?` that consumes a list of strings and produces `true` if all of the strings are equal, and `false` otherwise.

```
(strings-equal? empty) => true
(strings-equal? (cons "cs" (cons "cs" empty))) => true
(strings-equal? (cons "cs" (cons "se" (cons "cs" empty)))) => false
```

Hint: The template includes only one base case, but sometimes functions need multiple base cases.
Problem 3 - strings-equal? - Design Recipe

;; (strings-equal? los) Determines whether the strings in a list los are the same.
;; strings-equal?: (listof Str) → Bool
;; Examples:
(check-expect (strings-equal? (cons "sksk" (cons "sksk" empty))) true)

(define (strings-equal? los) . . .)

;; Tests:
(check-expect (strings-equal? (cons "VS" (cons "CO" empty))) false)
(check-expect (strings-equal? (cons "Girl" empty)) true)
(check-expect (strings-equal? empty) true)

Problem 4: Time Management

“So much to do, and so little time” How does one balance their personal and professional life?

Define a function that takes in a list of tasks, and produces a list of the weeks, days, and hours it will take to complete them all. (A task is any natural number in hours)

;; Examples:
(manage-time (cons 10 (cons 8 (cons 8 empty))))
→ (cons 0 (cons 1 (cons 2 empty)))
(manage-time (cons 168 (cons 51 empty)))
→ (cons 1 (cons 2 (cons 3 empty)))

Problem 4: Time Management - Design Recipe

;; (manage-time tasks) Consumes a lists of tasks and produces a list of the weeks, days, and hours required to complete them.  
;; manage-time: (listof Nat) → (listof Nat)
;; Example:
(check-expect (manage-time (cons 10 (cons 8 (cons 8 empty))))
  (cons 0 (cons 1 (cons 2 empty))))

(define (manage-time tasks) . . .)

;; Tests:
(check-expect (manage-time (cons 1000 empty)) (cons 5 (cons 6 (cons 16 empty))))
(check-expect (manage-time (cons 0 (cons 0 empty))) (cons 0 (cons 0 (cons 0 empty))))