Announcements

- Make sure to change your language level to Intermediate Student.

- Assignment 8 is due on **Tuesday, November 19, 9:00 PM**.

- The times and locations 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...

- Use functions as **First-Class Values**.
- Step through **Local** definitions.
- Use **Local** in your own functions.
- Understand the efficiency of **Local** definitions.
- Define your own **filter** function.
Review: Local Definitions

Recall the special form `local` which allows us to create local definitions.

The syntax for `local` is as follows:

```
(local [(define x1 exp1) . . . (define xn expn)] bodyexp)
```
Review: Local Benifits

Two reasons why we might want to use local expressions:

**Encapsulation** - local allows us to hide parts of our program from each other, since anything defined inside a local expression is not visible from outside the local. For example, we can define a helper function inside the function it is helping, and no function defined outside that main function have access to the helper.

**Efficiency** - We can use local variable to store the result of calculations to avoid recomputation.
Group Discussion: Efficiency with Local

(define (list-min lon)
  (cond [(empty? (rest lon)) (first lon)]
        [(<= (first lon)
            (list-min (rest lon)))]
        [(first lon)])
)

How can we use local constant definitions to improve efficiency?
Group Discussion: Efficiency with Local

(define (list-min-local lon)
  (cond [(empty? (rest lon)) (first lon)]
    [else (local [(define rest-min (list-min-local (rest lon)))]
      (cond [(<= (first lon) rest-min) (first lon)]
        [else rest-min]]))])

The recursive result is stored in a local constant and used twice.
Clicker Question 1: Local Definitions

(define (sum-lon alon)
  (local [(define (sum-lon-acc alon sum-so-far)
          (cond [(empty? alon) sum-so-far]
                [else (sum-lon-acc (rest alon)
                               (+ (first alon) sum-so-far)))]))]
  (sum-lon-acc alon 0)))

(sum-lon-acc (list 1 9 1 23) 0)

What would the following expression produce?
A) 1    B) 10    C) 11    D) 34    E) Error
Stepping Problem: Local

Provide a step-by-step evaluation of the following program. When renaming local definitions, append “_0” if possible, or else “_1”, “_2”, etc. Do not recopy any line that is already in its simplest form.

```
(define (f x y)
  (local [(define a (+ x 3))
    (define y 4)
    (define (g x) (+ x a))]]
  (* 2 (g y)))))
(f 7 3)
```
Clicker Question 2: Contracts with Function Types

What would be the contract for this function?

```
(define (f a b)
  (local [(define (f c) (+ (* a c) (* b c)))] f))
```

A  ;; f: Num Num → Num → Num
B  ;; f: Num Num → Num
C  ;; f: Num Num → (Num → Num)
D  ;; f: Num Num → Function
E  ;; f: Num Num → (fn Num → Num)
Review: filter

filter is a built-in function that takes in a list and a predicate, and will produce a new list keeping all the elements that satisfy the predicate function.

Here are some examples:

(filter even? '(1 2 3 4 5)) ⇒ ?
(filter string-alphabetic? '("pikachu" "cs135" "cs246" "mc") ) ⇒ ?
(filter string-lower-case? '("CS" "Math" "econ") ) ⇒ ?
Problem 1: filter-not

Define a function that does the exact opposite of filter, which removes all elements of a list that do satisfy the predicate function. One solution should utilize simple recursion, the other solution should not use explicitly recursion.

**Hint:** Utilize local definitions and the built-in filter function

(filter-not number? '(l o v 3)) ⇒ '(l o v)

(filter-not symbol? '(l o v 3)) ⇒ '(3)
Problem 1: Design Recipe

;; (filter-not pred lst) Produces a filtered version of lst
;; with all elements that satisfy predicate pred removed.
;; filter-not: (X → Bool) (listof X) → (listof X)
;; Example:
(check-expect (filter-not even? '(1 2 3 4 5)) '(1 3 5))

(define (filter-not pred lst) . . . )

;; Test:
(check-expect (filter-not posn? '()) '())
Problem 2: Average

The Math Faculty of Waterloo regularly holds a faculty dinner for students in Dean’s Honours list (cumulative average $\geq 87\%$). Here are some terminologies for average grade:

**MAV:** Major average

**CAV:** Cumulative average

**SMAV:** Special major average

Following is the data definition:

```
(define-struct student (name mav cav smav))
;; A Student is a (make-student Str Nat Nat Nat)
;; requires: mav, cav, smav $\leq 100$
```
Problem 2.1: Filter Cumulative-Average

Write a function `dean-honours-list` which consumes a list of `Students` and produces a list of `Students` who will be invited to the dinner (with a cumulative average $\geq 87\%$). You must use the built-in function `filter`. We encourage you to encapsulate your predicate functions in a `local` block with purposes and contracts.

```scheme
(define student-list (list (make-student "Ann" 89 86 90)
                          (make-student "John" 74 80 78)
                          (make-student "Bob" 93 95 90)))
(dean-honours-list student-list) =>
  (list (make-student "Bob" 93 95 90))
```
Problem 2.1: Design Recipe

;; (dean-honours-list students) Produce the list of students that
;; can be in dean-honours-list from the given list students.
;; dean-honours-list: (listof Student) → (listof Student)
;; Example:
(check-expect (dean-honours-list student-list)
              (list (make-student "Bob" 93 95 90)))

(define (dean-honours-list students) . . . )

;; Test:
(check-expect (dean-honours-list empty) empty)
Problem 2.2: Flixible Filter

The Math Faculty would like to make its student invitation software more flexible.

Write a function `honours-list` that consumes a selector accessor used to choose which average is used for selection (any of mav, cav, smav), and produces the list of students whose accessed average is greater than or equal to 87.

```
(honours-list student-list student-smav) ⇒
             (list (make-student "Ann" 89 86 90)
             (make-student "Bob" 93 95 90))
```
Problem 2.2: Design Recipe

;; (honours-list students accessor) Produce the honours-list of
;; students that accessed average is ≥ 87 from given list students.
;; honours-list: (listof Student) (Student → Nat) → (listof Student)
;; Example:
(check-expect (honours-list student-list student-smav)
  (list (make-student "Ann" 89 86 90)
        (make-student "Bob" 93 95 90))

(define (honours-list students accessor) . . . )
Problem 2.3: Flexible Filter & Average

Other faculties are jealous of Maths honor-list system and would like to create their own custom versions. They have their own selection thresholds and different cutoff grades.

Write `make-honours-list` that consumes a selector `accessor` and a minimum value `cutoff` and produces a the list of students who meet the minimum requirements of selection.

```
(make-honours-list student-list student-smav 80) ⇒
(list (make-student "Ann" 89 86 90)
  (make-student "Bob" 93 95 90))
```
Problem 2.3: Design Recipe

;; (make-honours-list students accessor cutoff) Produce
;; the list of students whose selected average
;; meets the cutoff from given list students.

;; make-honours-list:
;; (listof Student) (Student → Nat) Nat → (listof Student)

;; Example:
(check-expect (make-honours-list student-list student-mav 90)
  (list (make-student "Bob" 93 95 90)))

(define (make-honours-list students accessor cutoff) . . . )