Module 10: Imperative Programming, Modularization, and The Future

Throughout this course we have been using the *functional programming* paradigm.
Function programming:

> *"treats computation as the* **evaluation of mathematical functions** *and avoids changing-state and mutable data."*

```
(define (factorial n)
  (cond
   [(= n 0) 1]
   [else
    (* n
       (factorial (- n 1)))]))
```

Our function returns either a constant or the result returned by some function.

By now this may be so natural that it seems strange to mention.
But this is not the only way to approach program design!

An alternative way of thinking about program design is the *imperative programming* paradigm.
Imperative programming:

> *"uses statements that change a program's state, [and] consists of* **commands for the computer to perform**.*"*

```
def factorial(n):
    product = 1
    while n > 0:
        product = product * n
        n = n - 1

    return product
```

In this Python program, I instruct the computer to store a value in a new variable `product`. I then instruct the computer to repeatedly change these values, while some condition is true.

The function returns a value determined by following a series of instructions.

Language does not dictate paradigm

This course has been in Racket; it has followed the functional paradigm.
CS 116 is in Python; sections of it are in the functional paradigm, but other sections are in the imperative paradigm.

The choice of language does not dictate the paradigm. Some languages may make one paradigm easier to follow than the other.

- Racket supports functional programming very well, but has a full suit of instructions that support imperative programming: loops using `for` and `while`, variable re-assignment using `set!`, etc. **These are not part of this course.**
- Python supports imperative programming very well, but has many tools that support functional programming: `map`, `filter`, `lambda`, `functools.reduce`, and recursion.

Different programming paradigms are better in different contexts. Programmers who are familiar with multiple paradigms will be better programmers.

Moving toward imperative programming: `local`

Consider a simple function that calculates the average of a list of numbers:

```
;; (mean X) return the mean of X.
;; mean: (listof Num) -> Num
(define (mean X) (/ (foldr + 0 X) (length X)))
```

If I have a list I can operate on it and store the result in a constant:

```
(define my-list (list 4 6 20))
(define my-mean (mean my-list))
```

But so far, we cannot create a new constant *within* a function.
We cannot yet make constants that depends on the parameters our functions consume.

Example: Standard Deviation

Standard deviation is a measure widely used in statistics. The standard deviation σ of a set of N values $\{x_1, x_2, \ldots, x_N\}$, where μ is the mean of the values in X, is given by

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

I can write a function to calculate this, using `mean`:

```
;; (std X) return the standard deviation of X.
;; std: (listof Num) -> Num

(define (std X)
  (sqrt
   (/
    (foldr + 0
           (map (lambda (x) (sqr (- x (mean X)))) X))
    (length X))))
```

Example: Standard Deviation

```
;; (std X) return the standard deviation of X.
;; std: (listof Num) -> Num

(define (std X)
  (sqrt
   (/
    (foldr + 0
           (map (lambda (x) (sqr (- x (mean X)))) X))
    (length X))))
```

This function will work properly.

But there is still a problem with it: the computer needs to re-compute the same `(mean X)` many times, once for each item in `X`. In this course we do not discuss **algorithmic efficiency**, but you can imagine that re-calculating something is wasteful.

Wouldn't it be great if we could compute the mean **once**, and **store the result in a constant**?

Storing partial work with `local`

Using the **`local`** keyword we can compute one or more values and store the results in constants. The value of each constant can depend on the the parameters of the function, or on previously defined local constants.

`local` has two parameters:

1. In square brackets `[...]`, zero or more **definitions**.
2. An **expression** that may use these definitions. **`local`** evaluates to the value of this expression.

```
(define (my-fun n)
  (local
    [; local definitions
     (define local-var ...)
     ]
    ... ; an expression
    ))
```

Storing partial work with `local`

```
;; (sum-evens-minus-sum-odds L) return the sum of the even values in L
;;   minus the sum of the odd values.
;; sum-evens-minus-sum-odds: (listof Int) -> Int
(define (sum-evens-minus-sum-odds L)
  (local [
     (define odds  (filter odd? L))       ; store odd numbers
     (define evens (filter even? L))      ; store even numbers
     (define sum-odds  (foldr + 0 odds))  ; store sum of odds
     (define sum-evens (foldr + 0 evens)) ; store sum of evens
     ]
    (- sum-evens sum-odds) ; expression of final answer.
    ))
```

This code does very little work in the final expression.
Almost all the work is done in defining the constants.

Storing partial work with `local`

It is not necessary to do *any* computation in the final expression; it could simply be a constant!

```
;; (sum-evens-minus-sum-odds L) return the sum of the even values in L
;;   minus the sum of the odd values.
;; sum-evens-minus-sum-odds: (listof Int) -> Int
(define (sum-evens-minus-sum-odds L)
  (local [
     (define odds  (filter odd? L))       ; store odd numbers
     (define evens (filter even? L))      ; store even numbers
     (define sum-odds  (foldr + 0 odds))  ; store sum of odds
     (define sum-evens (foldr + 0 evens)) ; store sum of evens
     (define final-answer (- sum-evens sum-odds))
     ]
     final-answer)) ; just return a constant!
```

Exercise

Write a function `(normalize L)` that consumes a `(listof Num)`, and returns the list containing each value in `L` divided by the sum of the values in `L`.

Compute the sum only once.

`(normalize (list 4 2 14)) => (list 0.2 0.1 0.7)`

Example: Standard Deviation

This lets us rewrite `std` so it computes the mean of `X` just once, storing it in the constant `mu`.

```
;; (std X) return the standard deviation of X.
;; std: (listof Num) -> Num

(define (std X)
  (local [
          (define mu (mean X)) ;  a constant that stores the mean of X.
           ]
    (sqrt
     (/
      (foldr + 0
             (map (lambda (x) (sqr (- x mu))) X))
      (length X)))))
```

Example: Standard Deviation

...Or rewrite the function so it does things step-by-step, storing values as it goes along:

```
;; (std X) return the standard deviation of X.
;; std: (listof Num) -> Num

(define (std X)
  (local [
          (define mu (mean X))
          (define (sub-mu x) (- x mu))
          (define differences (map sub-mu X))
          (define sqr-differences (map sqr differences))
          (define sum-sqr-differences (foldr + 0 sqr-differences))
          (define N (length X))
          (define sum-over-N (/ sum-sqr-differences N))
          (define std-answer (sqrt sum-over-N))
          ]
    std-answer
    ))
```

Practice with `local`

Exercise

Write a function `(sum-odds-or-evens L)` that consumes a `(listof Int)`. If there are more evens than odds, the function returns the sum of the evens. Otherwise, it returns the sum of the odds.

Use **`local`**, but do not use `L` more than *twice* (in **`map`**, **`filter`**, **`foldr`**, or otherwise).

```
(sum-odds-or-evens (list 1 3 5 20 30)) => 9
```

`local` functions

We are not limited to only defining constants inside `local`. We can also use it to define **local functions**.

This is very similar to `lambda`, but allows us to name our functions.

```
;; (keep-multiples n L) return all values in L which are divisible by n.
;; keep-multiples : Nat (listof Nat) -> (listof Nat)
;; Examples :
(check-expect (keep-multiples 7 (list 2 3 5 28 7 3 14 77)) (list 28 7 14 77))

(define (keep-multiples n L)
  (local [
          ;; (divisible-n? x) return #true if x is divisible by n, else #false.
          ;; divisible-n?: Nat -> Bool
          (define (divisible-n? x)
            (= 0 (remainder x n)))
          ]
    (filter divisible-n? L)))
```

! Complete the design recipe for all `local` functions. Omit tests and examples.

local functions

Exercise

Create a function `(even-mean-minus-odd-mean L)` that returns the mean of the even values in `L` minus the mean of the odd values.

Include a **`local`** helper function `(mean M)` that consumes a `(listof Int)` and returns the mean of the values in `M`. Do not create any additional helper functions.

```
(even-mean-minus-odd-mean (list 16 14 5 1)) => 12
```

Encapsulation and Modularity

Another use of `local` is *encapsulation*. It allows us to create a helper function that we can use inside our primary function, but which is not available outside the primary function.

For example, suppose we had created a `mean` function as before, but in some context we wanted another function called `mean`, that did something else. We can simultaneously have a global function and a local function with the same name.

```
;; (mean L) return the mean of the values in L
(define (mean L) (/ (foldr + 0 L) (length L)))

;; (rms-fun L) do something with the rms-mean of L
(define (rms-fun L)
  (local [;; (mean L) return the rms-mean of the values in L
          ;; mean: (listof Num) -> Num
          (define (mean L) ; when I say mean, I mean root-mean-squared!
            (sqrt (foldr + 0 (map sqr L))))
          ]
    (* 20 (mean L))))
```

Summary: uses for `local`

There are several ways that **`local`** can be useful. Using it we can

- avoid re-computing the same value, making our code more efficient;
- use a named function instead of a **`lambda`**, potentially making our code easier to read;
- name a part of a computation, again making our code easier to read;
- let us encapsulate and modularize our code. We can make helpers that are not expected to be used outside a specific function.

These changes start us shifting away from a pure functional programming paradigm. You will become more fluent at imperative programming when you take CS116.

Module summary

- Be able to use `local` to define local constants, based on the values of parameters of your function.
- Be able to use `local` to define local functions, which are usable only inside the main function.
- Start to think **imperatively**: consider functions as a series of steps that the computer performs, one after the other.

Further Reading: *How to Design Programs* 16.2

Wrapping up CS115

I want to highlight:

- The importance of communication, to other programmers but even to yourself.
- That data definitions can guide the design of our programs.

These big ideas transcend language. They will influence your work in CS116 and beyond.

Looking ahead to CS116

CS116 uses Python. The syntax of this language is less restricted, which means

- There are more ways to say things (which makes things easier to say!)
- There are more ways to say things (there is more syntax to learn)

We will work on

- Writing programs that are longer, or deal with large amounts of data
- Designing programs so they run efficiently

Options for continuing CS

CS major:

- Take CS 116 and then CS 136.
- Talk to a CS advisor about the process.
- Many students have been successful in CS after starting in CS 115.

Computing Option:

- Take CS 116.
- Talk to an advisor to understand your choices.

Note: Participate in course selection! Otherwise, you may not be able to enrol in your preferred courses.