Module 1: Review and Preparations

Topics:

- What must you know from CS115?
- What will you do in CS116?

Readings: HtDP 1-20

Major Themes from CS115

- Design
- Common Patterns
- Verification
- Communication

CS115 was not a course *just* about Scheme!

Major Themes for CS116

- Design
- Common Patterns
- Verification
- Communication
- Algorithms

CS116 is not *just* a course about Python!

Review: Design Recipe

- Data Analysis and Design
- Determine needed functions
- For each function, write function specification
  - choose meaningful names
  - write contract and header
  - write purpose
- Examples

Design Recipe (continued)

- Body
  - choose an appropriate template
  - complete and or change template as needed
- Testing
  - include examples and other well-chosen test cases
  - compare expected answers to actual values produced by program
  - revise code as needed

Design Recipe (continued)

Program design still involves creativity, but the design recipe can be very helpful:

- It provides a place to start.
- Contracts and purpose can reduce simple syntax errors.
- Good design and template choices can
  - reduce logical errors
  - provide better solutions

Review: Structures

- Related data values in a single type
- Requires:
  - Structure definition
  - Data definition
- Scheme provides:
  - Constructor function
  - Selector functions
  - Type Predicate function

Structure: `residence`

Example: A residence has three features: the interior area (in square metres, including all floors) , the number of floors, and the number of people who live there.

How many square metres does an occupied residence have for each occupant?

Apply the Design Recipe

First step: Data Analysis

```
(define-struct residence
    (area floors occupants))
```

;; A **residence** is a value
;; **(make-residence a f oc),** where
;; **a** is a positive number, for the area in sq metres
;; **f** is a positive integer, for the number of floors
;; **oc** is a natural number, for the number of occupants

Next Step: Specification (write header, purpose and contract)

```
;; square-metres-per-occupant:
;;        residence -> num[>0]
;; computes the number of square metres of
;; space per resident in an occupied
;; residence r

(define (square-metres-per-occupant r) ..)
```


Add some examples

```
;; (square-metres-per-occupant
;;  (make-residence 80 2 1)) => 80

;; (square-metres-per-occupant
;;  (make-residence 300 20 2)) => 150
```


Choose a template and fill in the ...

```
;; (define (f r)
;;   (... (residence-area r) ...
;;    ... (residence-floor r) ...
;;    ... (residence-occupants r)
;;          ...))

(define (square-metres-per-occupant r)
  (/ (residence-area r)
     (residence-occupants r)))
```

Test the function

- Start with the examples:

```
(check-expect (square-metres-per-occupant
       (make-residence 80 2 1)) 80)
(check-expect (square-metres-per-occupant
       (make-residence 300 20 2)) 150)
```

- Add more cases if needed.

Review: Recursive Data

Requires a recursive data definition, including *at least* one base case and *at least* one recursive case

Example:

A **neighbourhood** is either

- **empty**, or
- **(cons r n)**, where **r** is a **residence**, and **n** is a **neighbourhood**.

Example: How many people live in a neighbourhood?

Specification

```
;; people-in-neighbourhood:
;;      neighbourhood -> nat
;; Produces the total number of
;; people in all of the residences
;; in neighbourhood n
(define (people-in-neighbourhood n)
```

Add some examples

```
;; (people-in-neighbourhood empty)
;;   => 0
;; (people-in-neighbourhood (list
;;     (make-residence 100 2 10)
;;     (make-residence 800 4 15))
;;   => 25
```


Choose a Template

```
(define (f n)
  (cond
    [(empty? n) ...]
    [else ... (first n)
          ... (f (rest n)) ...]))
```


Modify the Template

```
(define (people-in-neighbourhood n)
  (cond
    [(empty? n) 0]
    [else (+ (residence-occupants
                   (first n))
             (people-in-neighbourhood
                 (rest n)))]))
```

Testing

```
(check-expect (people-in-neighbourhood
  empty) 0)
(check-expect (people-in-neighbourhood
     (list (make-residence 100 2 10))) 10)
(check-expect (people-in-neighbourhood
     (list (make-residence 100 2 10)
           (make-residence 800 4 15)) 25)
(check-expect (people-in-neighbourhood
     (list (make-residence 100 2 0)
           (make-residence 800 4 0)) 0)
```


Warning: Design Recipe in Lecture Examples

- We may not always include full design recipe.
- It is still important!!!
- It is still required (in full) on assignments.

Review: Boolean Values

- Values: **`true`**, **`false`**
- Operations: **`or`**, **`and`**, **`not`**
  - **`(or x y ... z) =>`** **`true`** only if at least one of **`x,y,...,z`** is **`true`**
  - **`(and x y ... z) =>`** **`true`** only if all of **`x,y,...,z`** are **`true`**
  - **`(not x) =>`** **`true`** if x is **`false`**, otherwise **`false`**

Review: use of `local`

Use `local` to define constants and functions to be used inside a local expression

```
(local
    [(define ...)
     (define ...) ...]
    ( ... ))
```


Why use `local`?

- Readability
- Efficiency
- Encapsulation

Example: Find the maximum in a non-empty list of numbers.

First: Solve without using `local`

Recall that allowing only nonempty lists changes the function's base case:

A *nonempty list of numbers* is

- **`(cons n empty)`** where **`n`** is a number, or
- **`(cons n nel)`** where **`n`** is a number and **`nel`** is a *nonempty list of numbers*.

Specification & Template

```
;; list-max: (listof num)[nonempty] -> num
;; Produces the largest value in nel
;; Example: (list-max (list 2 -3 9)) => 9
(define (list-max nel)
   (cond
     [(empty? (rest nel))
           ... (first nel) ... ]
     [else ... (first nel) ...
           (list-max (rest nel)) ...]))
```


Complete the body

```
(define (list-max nel)
 (cond
   [(empty? (rest nel))
       (first nel)]
   [(> (first nel)
       (list-max (rest nel)))
    (first nel)]
   [else (list-max (rest nel))]))
```


Correct, but ...

- Very, very slow for some lists
- Count # times `list-max` is called for
  - **`(list 1 2 3)`**
  - **`(list 1 2 3 4 5 6 7 8 9 10)`**
  - **`(list 1 2 3 4 5 6 7 8 9 10 ... N)`**, for any positive integer **N**
- Recursive calls are repeated – duplicated work
- Exponential Growth in number of calls

Use `local` instead

```
;; list-max2: (listof num)[nonempty] -> num
(define (list-max2 L)
  (cond [(empty? (rest L)) (first L)]
        [else
         (local
           [(define rest-max
                    (list-max2 (rest L)))]
           (cond
             [(> (first L) rest-max)
                 (first L)]
             [else rest-max]))]))
```


How did this help?

- Count # times **`list-max2`** is called for
  - `(list 1 2 3)`
  - `(list 1 2 3 4 5 6 7 8 9 10)`
  - `(list 1 2 3 4 5 6 7 8 9 10 ... N)`,
    for any positive integer **N**
- Linear growth in number of calls

Review: Abstract List Functions

```
(define (square-list
                L)
 (cond
   [(empty? L) empty]
    [else
     (cons
      (sqr (first L))
     (square-list
      (rest L)))]))
```

```
(define (negate-list
                L)
 (cond
   [(empty? L) empty]
   [else
     (cons
      (not (first L))
     (negate-list
      (rest L)))]))
```

Note the similarities

- Both follow basic list template
- Both produce **`empty`** when **`empty`** consumed
- Both apply some function to first in list
- Both recursively build rest of list

➔Pass the function as a parameter!

➔Built-in function **`map`**

map

```
;; produces a list of values
;; created by applying f to
;; each value in lst
(define (map f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (map f (rest lst)))]))
```


Using map

```
(define (square-list lon)
      (map sqr lon))

(define (negate-list lob)
      (map not lob))

(define (lengths-list los)
      (map string-length los))
```

filter

```
;; produces a list of those values
;; in lst which produce true when f
;; is applied to them.
(define (filter f lst)
  (cond
    [(empty? lst) empty]
    [(f (first lst)) (cons (first lst)
              (filter f (rest lst)))]
    [else (filter f (rest lst))]))
```


Using filter

```
(define (even-elements loi)
   (filter even? loi))

(define (multiples-of-3 loi)
  (local
    [(define (mult-of-3? n)
       (zero? (remainder n 3)))]
    (filter mult-of-3? loi)))
```


foldr

```
;; produces the result of applying
;; combine successively through lst,
;; or base if lst is empty.
(define (foldr combine base lst)
  (cond
    [(empty? lst) base]
    [else (combine (first lst)
             (foldr combine
                base (rest lst)))]))
```

Using **foldr**

```
(define (sum-all lon)
  (foldr + 0 lon))

(define (concat-all los)
  (foldr string-append "" los))

(define (char-count los)
  (local
    [(define (add-chars-to s total-rest)
       (+ (string-length s) total-rest))]
    (foldr add-chars-to 0 los)))
```


Another Example: **longest-song**

```
(define-struct song (name length))
;; A song is a struct (make-song n l)
;; where n is a string (name of song),
;;       l is a nat (length of song,
;;       in seconds)

;; longest-song: (listof song) -> nat
;; Produces the length of the longest
;; song in songs
(define (longest-song songs)
  ...)
```


Goals of Module 1

Remember core concepts from CS115:

- design recipe
- basic syntax and patterns for Intermediate Student Scheme, including `local`
- functions as parameters in abstract list functions