Recursive Functions

CS 151: Introduction to Computer Science I

Types: Strings

`String` includes any phrase of keyboard characters:
“what do you want for dinner”, “b398af3”, “*-Z-? #0 ()[]}"

```
"Hello world!"
```

```
(:  is-it-5 :  Integer -> String)
;; tells you if the input is 5
(define (is-it-5 n)
  (if (= n 5)
      "It's 5!!"
      "Not 5..."))
```

Types: Strings

Compare Strings with string=?

```
(:  whats-for-breakfast :  String -> String)
(define (whats-for-breakfast desire)
;; respond to a breakfast request
  (if (string=?  desire "yogurt")
      "Coming right up!"
      "We don't have that..."))
```

Stick together Strings with string-append

```
(string-append "John&" "Paul&" "Ringo&" "George.")
⟹ "John&Paul&Ringo&George."
```

Types: Symbols

`Symbol` includes one-word phrases of keyboard characters:
'red, 'Chicago, 'computer-science, 'asleep

```
(: hunger : Real -> Symbol)
(define (hunger hours-since-eating)
  (if (> hours-since-eating 4.5)
      'very-hungry
      'kinda-hungry))
```

```
(: num-to-color : Integer -> Symbol)
(define (num-to-color n)
;; color the integers R, G, B, R, G, B, ...
  (if (= (remainder n 3) 0)
      'red
      (if (= (remainder n 3) 1)
          'green
          'blue)))
```

Factorial

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

$$= \begin{cases} n \cdot (n-1)! & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

```
(:  fac :  Integer -> Integer)
;; compute n!
(define (fac n)
  (if (= n 1)
      1
      (* (fac (- n 1)) n)))
```

Fibonacci

$$F_0 = 0, \qquad F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

```
(:  fibo :  Integer -> Integer)
;; compute the n-th Fibonacci number
(define (fibo n)
  (if (= 0 n)
      0
      (if (= 1 n)
          1
          (+ (fibo (- n 1)) (fibo (- n 2))))))
```

Tips on writing functions

- Write down the function's type.
- Do you think you may need to use recursion? Can you find a recursive definition for the thing you're trying to compute?
- What are the "base cases" or easy cases?
- Pick a few inputs, and imagine trying to evaluate the function on those inputs. How would you do it?
- Try writing your tests first!

Sum Numbers

Write a function `sum-a-few` that adds up the first n numbers.

```
(sum-a-few 6) ⟹ (+ 1 2 3 4 5 6) ⟹ 21
```

Recursive approach: first add 1 through $n - 1$, then add n

```
(:  sum-a-few :  Integer -> Integer)
;; adds up the first n natural numbers
;; requires the input isn't negative
(define (sum-a-few n)
  (if (= n 0)
      0
      (+ (sum-a-few (- n 1)) n)))
```

Character Counting

Given a `String`, count how many times the letters ‘c’ or ‘s’ appear.
Helpful functions:

```
(: string-length : String -> Integer)
;; return the number of characters in the input

(: string-ref : String Integer -> Char)
;; get the character at a given position
;; Note that the index starts at 0.
;; Requires the index to be nonnegative,
;; and less than the string's length

(: substring : String Integer -> String)
;; return the substring from the given position
;; until the end
```

Character Counting

```
(:  count-cs :  String -> Integer)
;; counts the number of characters in
;; the input that are ‘c’ or ‘s’
(define (count-cs phrase)
  (if (= 0 (string-length phrase))
      0
      (+
       (if (or
            (char=?  #\c (string-ref phrase 0))
            (char=?  #\s (string-ref phrase 0)))
           1
           0)
       (count-cs (substring phrase 1)))))
```

Helper Functions

With longer pieces of code, sometimes you'll want to write smaller functions on the way.

```
(:  first-char-cs :  String -> Integer)
;; return 1 if the first character is 'c' or 's',
;; 0 otherwise
;; Used as a helper function in count-cs
(define (first-char-cs phrase)
  (if (or
       (char=?  #\c (string-ref phrase 0))
       (char=?  #\s (string-ref phrase 0)))
      1
      0))
```

Infinite Loops

```
(:  fact :  Integer -> Integer)
;; compute n!
(define (fact n)
  (* (fact (- n 1)) n))
```

Infinite Loops

```
(:  fact :  Integer -> Integer)
;; compute n!
;; Requires n to be positive
(define (fact n)
  (if (= n 1)
      1
      (* (fact (- n 1)) n)))
```

```
(fact -1) ⟹ ??
```

Error

Use the `error` function to signal to the user that something is undefined or illegal

```
(:  fact :  Integer -> Integer)
;; compute n!
;; Requires n to be positive
(define (fact n)
  (if (<= 0 n)
      (error "fact:  input not positive")
      (if (= n 1)
          1
          (* (fact (- n 1)) n))))
```

Raises a *runtime error*

Convention: the error message should begin with the function name

cond

Write a function that takes in the name of a country and outputs its population
Old style:

```
(:  population :  String -> Integer)
;; get a country's population
(define (population country)
  (if (string=?  country "United States")
      325000000
      (if (string=?  country "Mexico")
          127500000
          (if (string=?  country "Canada")
              36000000
              ...))))
```

cond

Write a function that takes in the name of a country and outputs its population
New style:

```
(:  population :  String -> Integer)
;; get a country's population
(define (population country)
  (cond
    [(string=?  country "United States") 325000000]
    [(string=?  country "Mexico") 127500000]
    [(string=?  country "Canada") 36000000]
    [else (error "population:  unknown country")]))
```

cond

A general outline for recursive functions with > 1 base case:

```
(:  my-rec-fun :  Integer -> Integer)
(define (my-rec-fun n)
  (cond
    [base-case-1?  result-of-base-case-1]
    [base-case-2?  result-of-base-case-2]
        ⋮
    [else recursive-case]))
```

```
(:  fibo :  Integer -> Integer)
(define (fibo n)
  (cond
    [(= n 0) 0]
    [(= n 1) 1]
    [else (+ (fibo (- n 1)) (fibo (- n 2)))]))
```

List

The type `(Listof Number)` represents a list of numbers:

$$[97, -1, 5, 12, 6]$$

Create something of this type with the `list` function:

```
(: my-list : (Listof Number))
(define my-list (list 97 -1 5 12 6))
```

A whole family of new types!
`(Listof Real)`, `(Listof Boolean)`,
`(Listof (Listof String))`

List

```
(: my-list : (Listof Number))
(define my-list (list 97 -1 5 12 6))
```

Basic list functions: `first`, `rest`, `empty?`

```
(first my-list) ⟹ 97
```

```
(rest my-list) ⟹ '(-1 5 12 6)
```

```
(empty? my-list) ⟹ #f
```

List recursion: sum

Given a list of Numbers, add them up.

```
(: sum-nums : (Listof Number) -> Number)
;; sums the numbers in the list
(define (sum-nums nums)
  (cond
    [(empty? nums) 0]
    [else (+
           (sum-nums (rest nums))
           (first nums))]))
```

Many other functions follow this pattern:

```
(: length : (Listof Number) -> Integer)
(: list-or : (Listof Boolean) -> Boolean)
(: append-all : (Listof String) -> String)
```

List recursion: counting

Given a list of `Strings` of titles of books you own, count how many copies of "Oedipus Rex" you have.

```
(: count-occurrences : (Listof String) String ->
Integer)
(: count-oedipus : (Listof String) -> Integer)
```

List recursion: counting

```
(: count-occurrences : (Listof String) String ->
Integer)
;; counts the number of times that title
;; appears in my-books
(define (count-occurrences my-books title)
  (cond
    [(empty? my-books) 0]
    [else (+
           (count-occurrences (rest my-books)
title)
           (if (string=? (first my-books) title) 1
0)]))

(: count-oedipus : (Listof String) -> Integer)
(define (count-oedipus my-books)
  (count-occurrences my-books "Oedipus Rex"))
```

Selling Coconuts

You're selling coconuts. Each time you sell a coconut, you raise the price by $3. Initially, the price is $1. If *n* people buy coconuts, how much will you make?

```
(: profit : Integer -> Integer)
```

```
(: price : Integer -> Integer)
(define (price i)
;; get the price of the i-th coconut sale
  (if (= 1 i) 1 (+ 3 (price (- i 1)))))

(: profit : Integer -> Integer)
;; returns the profit from n coconut sales
(define (profit n)
  (if (= 0 n) 0
      (+ (profit (- n 1)) (price n))))
```

Selling Coconuts

Approach 2: we know the n-th person pays $3n + 1$. Add that to the total paid by the other $n - 1$ people

```
(:  profit :  Integer -> Integer)
;; returns the profit from n coconut sales
(define (profit n)
  (if (= 0 n)
      0
      (+ (profit (- n 1)) (+ (* 3 n) 1))))
```

Approach 3: after the first person buys, the price goes up by \$3. Your profit is the starting price plus how much you make from $n - 1$ people buying at a higher start price.

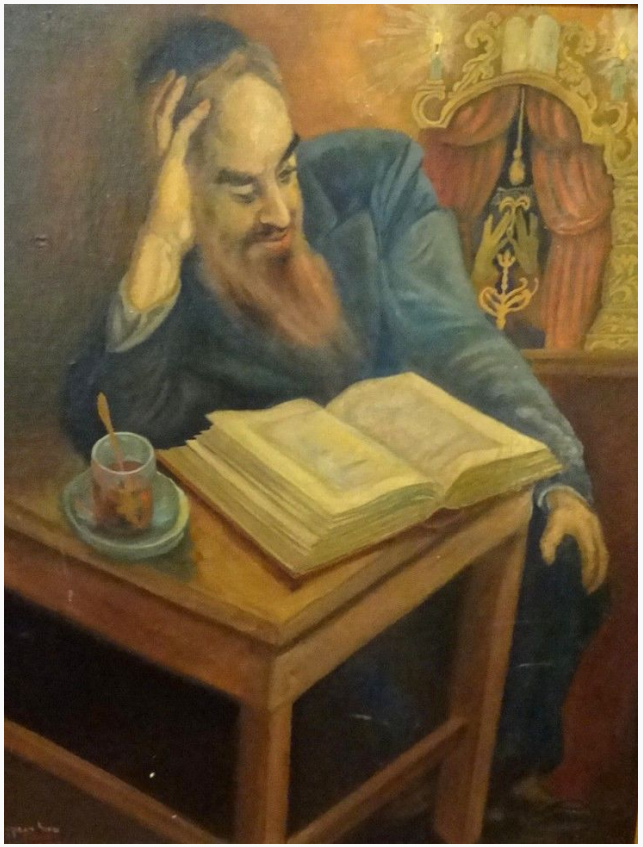
Selling Coconuts

```
(:  profit-from-start :  Integer Integer ->
Integer)
;; compute how much money is made from n people
;; when the price is initially 'start'
(define (profit-from-start n start)
  (if (= 0 n) 0
      (+
       start
       (profit-from-start (- n 1) (+ start 3)))))

(:  profit :  Integer -> Integer)
;; returns the profit from n coconut sales
(define (profit n)
  (profit-from-start n 1))
```

What to know

- Recursive functions
- `error` function
- `cond`
- list and Listof



Bring your laptop to today's lab!