Lecture #27: Scheme Examples

- A little philosophy: why are we talking about interpreters, etc.?

- Idea is to understand your programming language better by understanding common concepts in the design of programming languages

- ...And also to get better mental models of what programs are doing by actually studying how a program might be executed.

- With this, you can perhaps develop better intuitions about what usages are likely to be expensive.

- More directly, many projects can benefit from the introduction of specialized "little languages" and studying interpreters gives you some background in defining and implementing them.
Tail-Recursive Length?

Last time, we came up with this:

;; The length of list L
(define (length L)
  (if (eqv? L '()) ; Alternative: (null? L)
      0
      (+ 1 (length (cdr L)))))

but this is not tail recursive. How do we make it so?
Tail-Recursive Length: Solution

;;; The length of list L
(define (length L)
    ;; n + the length of R.
    (define (length+ n R)
        (if (null? R) n
            (length+ (+ n 1) (cdr R))))
    (length+ 0 L))
Standard List Searches: assoc, etc.

• The functions `assq`, `assv`, and `assoc` classically serve the purpose of Python dictionaries.

• An *association list* is a list of key/value pairs. The Python dictionary `{1 : 5, 3 : 6, 0 : 2}` might be represented

  `((1 . 5) (3 . 6) (0 . 2))`

• The `assx` functions access this list, returning the pair whose `car` matches a key argument.

• The difference between the methods is whether we use `eq?` (Python `is`), `eqv?` (more like Python `==`), or `equal?` (does “deep” comparison of lists).

  ```scheme
  ;; The first item in L whose car is eqv? to key, or #f if none.
  (define (assv key L)
  )
  ```
Assv Solution

;; The first item in L whose car is eqv? to key, or #f if none.
(define (assv key L)
  (cond ((null? L) #f)
        ((eqv? key (caar L)) (car L))
        (else (assv key (cdr L)))))

• Why caar?
  - L has the form ((key1 . val1) (key2 . val2) ...).
  - So the car of L is (key1 . val1), and its key is therefore (car (car L)) (or caar for short).
A classic: reduce

`; Assumes f is a two-argument function and L is a list.
`; If L is (x1 x2...xn), the result of applying f n-1 times
`; to give (f (f (... (f x1 x2) x3) x4) ...).
`; If L is empty, returns f with no arguments.
`; [Simply Scheme version.]
`; >>> (reduce + '(1 2 3 4)) ===> 10
`; >>> (reduce + '()) ===> 0
(define (reduce f L)
  )
Reduce Solution (1)

;; Assumes f is a two-argument function and L is a list.
;; If L is \((x_1 \ x_2...x_n)\), the result of applying \(f\) \(n-1\) times
;; to give \((f (f (\ldots (f x_1 x_2) x_3) x_4) \ldots)\).
;; If L is empty, returns \(f\) with no arguments.
(define (reduce f L)
  (cond ((null? L)
         (f)) ; Odd case with no items
        ((null? (cdr L))
         (car L)) ; One item
        (else
         (reduce f (cons (f (car L) (cadr L))
                       (cddr L))))
  )
; E.g.:
;   (reduce + '(2 3 4))
;   -calls-> (reduce + (5 4))
;   -calls-> (reduce + (9))
;   -yields-> 9
Reduce Solution (2)

;; Assumes f is a two-argument function and L is a list.
;; If L is (x1 x2...xn), the result of applying f n-1 times
;; to give (f (f (... (f x1 x2) x3) x4) ...).
;; If L is empty, returns f with no arguments.
(define (reduce f L)
    (define (reduce-tail accum R)
        (cond ((null? R) accum)
            (else (reduce-tail (f accum (car R)) (cdr R))))
    (if (null? L) (f) ;; Special case
        (reduce-tail (car L) (cdr L))))