Lecture 28: Scheme and Interpretation

Controlling Function Evaluation

- The standard function `apply` has the effect of allowing one to construct and evaluate function calls.
- To call a function, one generally needs to know how many arguments it takes, and then wire that into the call expression, as in `f(x,y)`—you may not know what precise function `f` is, but you must know how many arguments it takes.
- In Lisp (and Scheme) the function `apply` handles this:

```
(define L '(1 2 3))
   (apply + L) ===> (+ 1 2 3) ===> 6
```

- More recently, we see these in other programming languages. In Python, one can write `f(*L)` for `(apply f L)`.

Another classic: map

- Ignore that this is actually built-in.
- The obvious way goes like this:

```
;;; Assumes f is a one-argument function and L is the
;;; list (x1 ... xn).  Returns the list ((f x1) ... (f xn)).
(define (map1 f L)     ;; map1 to distinguish from full map.
    (if (null? L) '()
        (cons (f (car L)) (map1 f (cdr L)))))
```

- Two problems:
  1. Not tail recursive. [Hint: `reverse` is built in].
  2. How to do the full version: `(map f L1 ... Lm)`, where we compute `((f (car L1) ... (car Lm)) ...)`?

```
;;; Assumes f is a k-argument function and L is a non-empty list
;;; of equal-length lists, L1...Ln. Returns the list
;;; ((f x11 x21 ...) ... (f x1n ...)), (xij is item j of list i).
(define (map f . L)  ;; Like Python's def map(f, *L)
)
```


Solution: Tail-Recursive Map1

```
(define (map1 f L)
   ;; The reverse of (map1 f L) prepended to the list sofar.
   (define (map1-tail sofar L)
       (if (null? L) sofar
           (map1-tail (cons (f (car L)) sofar) (cdr L))))
   (reverse (map1-tail '() L))
)
```


Solution: Full Map

- Non-tail-recursive:

```
(define (map f . LL)
   (if (null? (car LL)) '()
       (cons (apply f (map1 car LL))
             (map f (cdr LL)))))
```

- Tail-recursive:

```
(define (map f . LL)
   ;; The reverse of (map f L) prepended to the list sofar.
   (define (map-tail sofar LL)
       (if (null? (car LL)) sofar
           (map-tail (cons (apply f (map car LL)) sofar)
                     (map-tail (apply cdr LL)))))
   (reverse (map-tail '() LL))
)
```


Eval

- From early on, Lisp systems have used the fact that programs simply data that is processed by an evaluator.
- The `eval` function has been in Lisp for some time.
- It treats its argument as a Lisp expression and evaluates it.
- E.g., `(eval (list + 1 2))` produces 3.
- Only recently added to Scheme officially (since version 5), perhaps in part because it is a little more difficult to define in Scheme than in original Lisp.
- One difficulty is that original Lisp was *dynamically scoped*, but Scheme (like Python) is *statically scoped.*

Static and Dynamic Scoping

- The scope rules are the rules governing what names (identifiers) mean at each point in a program.
- We've been using environment diagrams to describe the rules for Python (which are essentially identical to Scheme).
- But in original Lisp, scoping was *dynamic*.
- Example (using classic Lisp notation):

```
(defun f (x)    ;; Like (define (f x) ...) in Scheme
       (g))
(defun g ()
       (* x 2))
(setq x 3)      ;; Like set! and also defines x at outer level.
(g)             ;; ===> 6
(f 2)           ;; ===> 4
(g)             ;; ===> 6
```

- That is, the meaning of `x` depends on the most recent and still active definition of `x`, even where the reference to `x` is not nested inside the defining function.

Eval and Scoping

- Dynamic scoping made `eval` easy to define: interpret any variables according to their "current binding."
- But `eval` in Scheme behaves like normal functions, it would not have access to the current binding at the place it is called.
- To make it definable (without tricks) in Scheme, one must add a parameter to `eval` to convey the desired environment.
- In the fifth revision of Scheme, one had the choice of indicating an empty environment and the standard, builtin environment.
- Our STk interpreter goes its own way:
  - `(eval E)` evaluates in the global environment.
  - `(eval E (the-environment))` evaluates in the current environment.
  - `(eval E (procedure-environment f))` evaluates in the environment pointed to by function `f`.