Lecture #26: Announcements

Project 1 revisions: due April 10.

Hackathon: "There is a hackathon hosted by H@B—Big Hack—vs. Stanford this Weekend. Prizes include: Macbook Airs, Retina iPads, Occulus Rifts, Pebble Smartwatches. You don't need to have hackathon experience to attend; it'll be a lot of fun! This Weekend Saturday April 5th to Sunday April 6th (We'll be back in Berkeley by 6pm). Workshop today that will give you ideas and give you tips about hackathons from previous winners. Please visit the Big Hack facebook page for information on how to register and if you have any questions." – Apoorva Dornadula

Webcast Survey: "Educational Technology Services (ETS) is planning the future of the Webcast program; in particular we will be making choices about *where* we make course webcasts available to you. Your input is valuable! Please take a few minutes to answer the questions using the link [in the Announcements section on the home page.]"

Scheme and Scheme Interpretation

- 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.

A Bit More Scheme. 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 `ass`x 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?` (handles numbers better), or `equal?` (more like Python ==).

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

Assv

```
;; 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 (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)
    (cond ((null? L)
           (f))       ; Odd case with no items
          ((null? (cdr L))
           (car L))  ; One item: apply f 0 times
          (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))))
```

Another classic: (two-argument) 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 because it takes only one list
```

Another classic: (two-argument) 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 because it takes only one list
    (if (null? L) '()
        (cons (f (car L)) (map1 f (cdr L)))))
```

Tail-Recursive Map1

- Previous implementation is not tail recursive.
- **Hint:** `reverse` is built in.

```
;;; 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)



)
```

Tail-Recursive Map1

- Previous implementation is not tail recursive.
- **Hint:** `reverse` is built in.

```
;;; 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)
   ;; The reverse of (map1 f L) prepended to the list sofar.
   (define (map1-tail sofar L)

                                                          )
   (reverse (map1-tail '() L))
)
```

Tail-Recursive Map1

- Previous implementation is not tail recursive.
- **Hint:** `reverse` is built in.

```
;;; 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)
   ;; 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))
)
```

Problem: Unknown Argument List Lengths

- The full `map` function is like that in Python:

```
(map + '(1  2  3) '(10 20 30) '(100 200 300))
===> (list (+ 1 10 100) (+ 2 20 200) (+ 3 30 300))
===> (111 222 333)
```

- So there must be a step in `map` where we call its function argument with a *list* of parameters of arbitrary length.
- In Python, can do such things with `*`, as in `f(*L)`.
- In Scheme, it is a built-in function: `apply`.

Apply: 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
```

Eval

- From early on, Lisp systems have used the fact that programs are reprsented as Lisp 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 parent environment of function `f`.

Interpreters: Eval and Apply

- In a Scheme interpreter, one of the main activities of `eval` is evaluating calls.
- The intepreter sees the expression

  ```
  (f (+ 1 2) y)
  ```

  (which is Scheme data). What does it do?
- First, it *identifies* this as a call: it's a non-empty list, and its first item is not "special."
- So it (recursively) evaluates `f`, `(+ 1 2)`, and `y`, giving a function and two other values.
- Now it must call the function with those two arguments, for which it uses `apply`.
- If `f` is a non-primitive function (defined via `define` in the same program), then `apply` will eventually have to *evaluate* its body...
- ...for which it uses `eval`.
- And so it goes, back and forth between `eval` and `apply` until we get down to primitive functions and other base cases.