Lecture #27: Scheme Iteration

Public Service Announcement

"Bay Bithack - the first and largest collegiate bitcoin/blockchain hackathon in the world!
This weekend: April 2–3
No cryptocurrency experience required
Free food, swag, prizes, and more!
Register at baybithack.com"

Recursion and Iteration

- We've mentioned before that Scheme uses recursion where most other languages (such as Python) use special iterative constructs.
- This puts a special burden on Scheme interpreters to handle iterative recursions, known as *tail recursions* well.
- From the reference manual:

> "Implementations of Scheme must be *properly tail-recursive.* Procedure calls that occur in certain syntactic contexts called *tail contexts* are tail calls. A Scheme implementation is properly tail-recursive if it supports an *unbounded number of [simultaneously] active tail calls.*

- First, let's define what that means.

Tail Contexts

- Basically, an expression is in a *tail context* if it is evaluated last in a function body and provides the value of a call to that function.
- A function is *tail-recursive* if all function calls in its body that can result in a recursive call on that same function are in tail contexts.
- In effect, Scheme turns recursive calls of such functions into iterations by *replacing* those calls with one of the function's tail-context expressions instead of simply returning.
- This decreases the memory devoted to keeping track of which functions are running and who called them to a constant.

Tail Contexts in Scheme

- Tail contexts are defined inductively (or recursively). The "bases" are

  (lambda (*ARGUMENTS*) $EXPR_1$ $EXPR_2$ ... $EXPR_n$) ; Tail contexts in Blue
  (define (*NAME ARGMENTS*) $EXPR_1$ $EXPR_2$ ... $EXPR_n$)

  ('*EXPR*' means "Scheme expression")

- If an expression is in a tail context, then certain parts of it become tail contexts all by themselves:

  (if *EXPR* *THEN-EXPR* *ELSE-EXPR*)

  (cond ($COND\text{-}EXPR_1$ $EXPR_{11}$ $EXPR_{12}$ ... $EXPR_{1n}$)
  ($COND\text{-}EXPR_2$ $EXPR_{21}$ $EXPR_{22}$ ... $EXPR_{2n}$)
  ...)

  (and $EXPR_1$ ... $EXPR_n$)
  (or $EXPR_1$ ... $EXPR_n$)

  (begin $EXPR_1$ ... $EXPR_n$)

Tail-Recursive Length?

On several occasions, we've computed the length of a linked list like 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 ass$x$ functions access this list, returning the pair whose `car` matches a key argument.
- The difference between the methods is that
  - `assq` compares using `eq?` (Python **is**).
  - `assv` uses `eqv?` (which is like Python `==` on numbers and like **is** otherwise).
  - `assoc` uses `equal?` (does "deep" comparison of lists).

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

- This is a tail-recursive function.
- 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
          (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))))
```

A Harder Case: Map

- We've seen `map` many times.
- An obvious Scheme version:

```
;; Assuming f is a one-argument function and L a list, the list of
;; results of applying f to each element of L
(define (map f L)
    (if (null? L) ()
        (cons (f (car L) (map f (cdr L))))))
```

- Is this tail-recursive?

Making map tail recursive

- Need to pass along the partial results and add to them.
- Problem: `cons` adds to the *front* of a list, so we end up with a reverse of what we want.

```
(define (map f L)
     ;; The result of prepending the reverse of (map rest) to
     ;; the list partial-result
     (define (map+ partial-result rest)
         (if (null? rest) partial-result
             (map+ (cons (f (car rest)) partial-result)
                   (cdr rest))))
     (reverse (map+ () L)))
```

- What about `reverse`?

And Finally, Reverse

- Actually, we can use the very problem that `cons` creates to solve it!
- That is, consing items from a list from left to right results in a reversed list:

```
(define (reverse L)
    (define (reverse+ partial-result rest)
        (if (null? rest) partial-result
            (reverse+ (cons (car rest) partial-result)
                      (cdr rest))))
    (reverse+ () L))
```