Lecture 19: Scheme II

July 25, 2022

Richard Roggenkemper

Announcements

- Ants due 7/28!
  - Extra Credit for early Submission
  - Extra Credit Problem too!
- Magic: The Lambda-ing due 7/28!
  - Worth 2 Extra Credit Points
- Slightly different Instructor OH times this week
  - Always check cs61a.org/office-hours before going to OH!
- Have the Scheme Specification open when you are doing assignments

Review

Scoping

So far we have seen lexical scoping in Python and Scheme which means that the parent of a function is where the function is defined

In Scheme, we can also use a mu special form instead of lambda to get dynamic scoping where the parents is the frame where the function is called

Tail Recursion

A tail call is a call expression in a tail context

A function is tail recursive if all recursive calls are tail calls

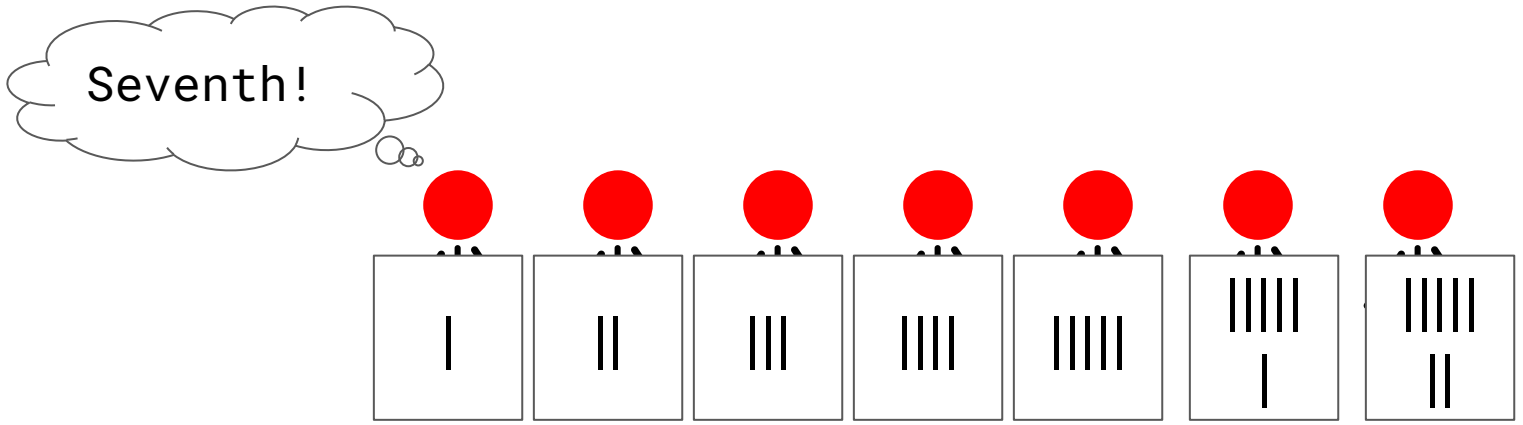
Functions in Scheme and Python can be Tail Recursive, but only Scheme has Tail Call Optimization which can help with the space efficiency of the function

This optimization can help bring a linear space use down to constant

Line Example 2: Electric Boogaloo

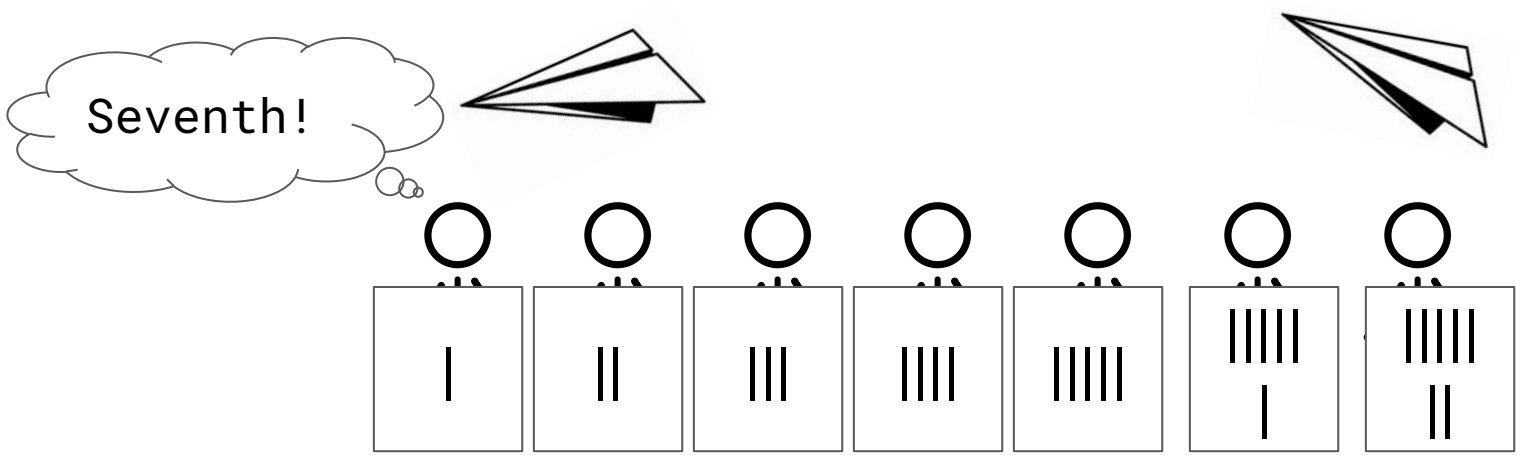
We are back at Crossroads and there is still a line. How would we solve the problem with Tail Recursion? What would Optimizing it look like?

1. Pass a piece of paper down
2. Every person marks it with a tally mark
3. When the first person marks it, we have the total!
4. Tell the person behind you the total until it gets to the person who started it



Optimized Example

1. Pass a piece of paper down
2. Every person marks it with a tally mark
3. When the first person marks it, we have the total!
4. Turn the paper into a paper airplane, and fly it back



Tail Recursive Sum

```
(define (sum lst)
      (define (helper lst so-far)
            (if (null? lst)
                  so-far
            (helper (cdr lst) (+ so-far (car lst)))))
         (helper lst 0))
```

Class Methods

@classmethod

By default, a function definition inside a class is a bound method that receives an instance of that class

To instead make a function that receives the class itself, use the @classmethod decorator

Instead of `def function(self, arg)`

We will use `def function(cls, arg)`

Most commonly used for “factory methods” where a method is used to construct a return a new instance of the class

Class Method Example

```
class Rabbit(Animal):
    species_name = "European rabbit"
    scientific_name = "Oryctolagus cuniculus"
    calories_needed = 200
    play_multiplier = 8

    @classmethod
    def rabbit_twins(cls, name1, name2):
        rabbit1 = cls(name1)
        rabbit2 = cls(name2)
        return [rabbit1, rabbit2]

twinsies = Rabbit.rabbit_twins(“Shoo Shoo”, “Buster”)
```

Scheme

Demo

Begin

begin lets you combine multiple expressions into one expression

This is useful when you only have one place but need multiple expressions (cond , if, etc.)

```
if x > 10:
    print('big')
    print('guy')
else:
    print('small')
    print('fry')
```

```
(if (> x 10) (begin
                (print 'big)
                (print 'guy))
             (begin
                (print 'small)
                (print 'fry)))
```

Let

Let special form binds symbols to values temporarily, so that after the expression, the binding is no longer there

```
a = 3
b = 2 + 2
c = math.sqrt(a * a + b * b)
a and b are still bound down here
```

```
(define c (let ((a 3)
                (b (+ 2 2)))
            (sqrt (+ (* a a) (* b b)))))
a and b are not bound down here
```

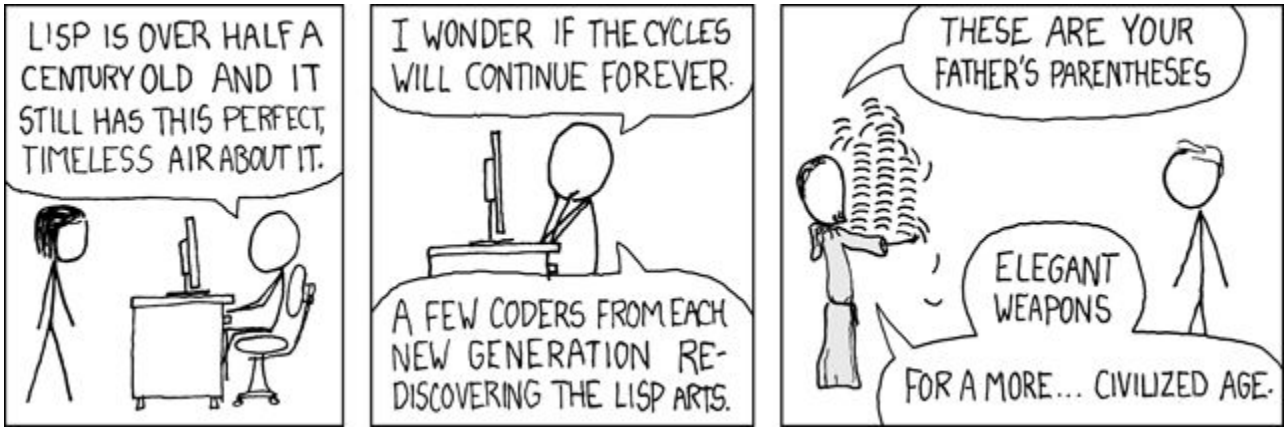
List Review

Scheme	**Python**
(cons a b)	Link(a, b)
(car a)	a.first
(cdr a)	a.rest
(list a b c)	Link(a, Link(b, Link(c)))

List Equivalence

In Python, we saw that `is` and `==` showed 2 different kinds of equivalence for lists

In Scheme, we have the same idea where we have 2 procedures, one to tell us if the lists have the same values and one to see if the lists are the same



equal?

The equal? procedure will tell us if the values of two lists are the same. Similar to == in Python

```
(define list1 '(a b c))
(define list2 '(a b c))
(define list3 list1)

(equal? list1 list2)
#t
(equal? list1 list3)
#t
```

Demo

eq?

The eq? procedure will tell us if the two lists are pointing to the same list. Similar to `is` in Python (both have 2 letters)

```
(define list1 '(a b c))
(define list2 '(a b c))
(define list3 list1)


(eq? list1 list2)
#f
(eq? list1 list3)
#t
```

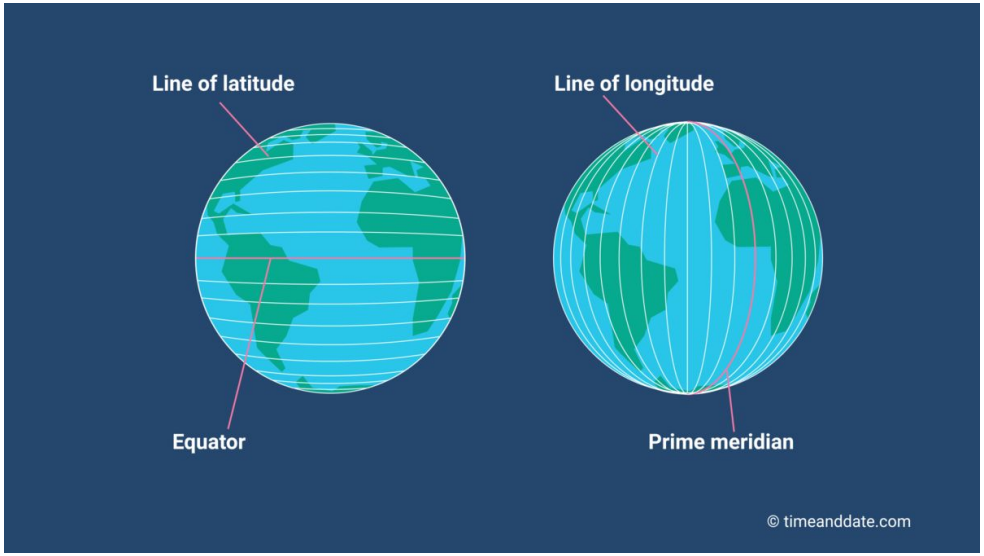
Demo

List Manipulation

length	(length ‘(1 2 3) ; expect 3
append	(append '(1) '(2 3)) ; expect (1 2 3)
map	(map abs '(-1 -2 3 4)) ; expect (1 2 3 4)
filter	(filter even? '(0 1 2 3 4 5)) ; (0 2 4)
reduce	(reduce + '(0 1 2 3 4 5)) ; 15

North of Equator

Implement (`north-of-equator point`), a procedure that takes a point, a two element list with a latitude and longitude , and returns whether point is north of the Equator



North of Equator Solution

```
(define (north-of-equator point)
        (> (car point) 0))
```

All North

Now, implement `(all-north points)`, a procedure that takes a list of points and returns whether all points are north of the Equator.

All North Solution

```
(define (all-north points)
      (= (length (filter north-of-equator points))
            (length points)))


(define (all-north points)
  (cond
   ((null? points) #t)
   ((north-of-equator (car points)) (all-north (cdr points))
   (else #f)))
```

Break

Programs as Data

Scheme Expressions

Expressions in Scheme Programs are either primitives or combinations

Primitives: `8` , `19.99`, `#f`, `*` , `if`

Combinations: `(if 8 19.99 #t)`, `(not #f)`

Combinations are just Scheme Lists which means I should be able to create them as lists

Demo

Scheme Eval

Scheme has a built-in procedure called **eval** which when called on a created combination will evaluate it

So now we have the ability to create combinations with Scheme Lists and the ability to call those combinations and evaluate them

Check Your Understanding

For the follow Scheme call expressions, create them as Scheme Lists

```
(print 8)
```

```
(+ 22 (* 20 10))
```

```
(if #t (print 'true) (print 'false)
```

Check Your Understanding

```
(print 8)
```

```
> (list ‘print 8)
```

```
(+ 22 (* 20 10))
```

```
> (list ‘+ 22 (list ‘* 20 10))
```

```
(if #t 3 100)
```

```
> (list ‘if #t 3 100)
```

Creating Lists

As you can see , with longer, more complicated expressions, creating them as lists might be really difficult

We learned of 3 ways to create lists (quote, cons, list) , but none of them really work well

The best one is quote, because we just type in what we want, but we don’t have the control of evaluating anything

With quasiquotes, we will add the ability to unquote (evaluate) certain elements which gives us more control

Demo

Quasiquotation

```
scm> '(a b)
(a b)
scm> `(a b)
(a b)
scm> (define b 4)
b
scm> '(a ,(+ b 1))
(a (unquote (+ b 1)))
scm> `(a ,(+ b 1))
(a 5)
```

Check Your Understanding

```
scm> (define x 5)
x
scm> (define y 10)
y
scm> `(x y 5)
???
scm> `(+ ,x ,y)
???
scm> `(if (> ,x ,y) #t #f)
???
scm> (eval `(if (> ,x ,y) #t #f))
???
```

Check Your Understanding

```
scm> (define x 5)
x
scm> (define y 10)
y
scm> `(x y 5)
(x y 5)
scm> `(+ ,x ,y)
(+ 5 10)
scm> `(if (> ,x ,y) #t #f)
(if (> 5 10) #t #f)
scm> (eval `(if (> ,x ,y) #t #f))
#f
```

Demo

Exponents with Lists

Write a function `(exp base n)` which returns a list, which returns the base to the n-th power (base ** n) when eval is called on it.

Think about how to incorporate the ideas that we have been talking about into this problem. You might not know how to solve it, but think about Scheme, recursion, and programs as data.

```
scm> (exp 5 1)
(* 1 5)
scm> (exp 5 2)
(* (* 1 5) 5)
```

Summary

- Scheme Lists have the same equivalence checks (different procedure names though) as Python
- We still have lots of list manipulation procedures in Scheme
- Scheme Expressions are either combinations or primitives
- Combinations are just Scheme Lists which can be evaluated