PROBLEM SOLVING 11

COMPUTER SCIENCE 61A

July 24, 2012

Today's section will be a kind of "Meta-Section", we are going to walk through some medium to hard-ish problems in Scheme, and we will discuss some methods to attack these problems

1 Problem Solving

1.1 Scheme

"Limitations" of Scheme

A lot of you may feel limited by scheme, since it lacks all the data structures and niceties you've been spoiled with in Python, but Scheme gives you all the tools you need to tackle most problems.

Recursion

At the heart of every Scheme problems is your favorite topic from 61A...recursion! Because Scheme doesn't have iteration in order to solve most problems you will need to recurse some how. And some times this takes some getting used to.

Getting Around the Syntax

Perhaps the Syntax of Scheme may throw a lot of you guys off. Well syntax does not make a program! If you can solve the problem in "Python Syntax", with the limitations that Scheme gives you, translating it to true Scheme syntax shouldn't be too difficult. If that's hard try writing the problem in full Python, and slowly try to ease away the things you can't do in Scheme until you reach code that can be easily translated to Scheme. You will find in most cases that the format of the particular program will not vary.

Let's walk through a problem now

1.2 Filter

Let's write a filter function! I'm sure you guys love this function from python. Lets write it together now so we can use it for our later problems.

```
(define (filter lst, pred)
;;your code here
)
```

Bells in your heads should be ringing recursion! If it isn't it's ok let's walk through this one. First lets think about what filter does. First of all you should ask yourself if it returns a new list, does it?

Well it does. So we know the cons will be used somewhere. But before we get ahead of ourselves, we know that filter walks through the entire list, so the logical way to do this in python would be a for loop. But we can't do that, so lets try writing this in python recursively.

The first thing we will do is define our base case, which is when our list is empty. If you have been noticing our base cases for recursive functions is usually when we take the smallest (or largest) possible input we can handle. In this case its the empty list. And an empty list is already filtered.

```
def filter(lst, pred):
        if lst == [ ]:
                return []
```

Now lets think about our recursive case. We know we want to return a list and we want to construct the list as we progress in our recursion. And we know we need to check our pred function with the current input some how. So what would the next step be?

```
def filter(lst, pred):
        if lst == [ ]:
                return []
        if pred(lst[0]):
                return [lst[0]] + filter(lst[1:], pred)
        return filter(lst[1:], pred)
```

Now if you notice carefully at the code, I haven't broken scheme conventions, I only looked at the first and rest of the list, so lets try converting this into scheme.

```
(define (filter lst pred)
        (if (null? lst)
        st
        (if (pred (car lst))  (cons (car lst) (filter (cdr lst) pred) )
        (filter (cdr lst) pred) )))
```

Notice how the two solutions (in python and scheme) have very similar forms. I changed the access to the first element to a call to car, changed the base case check to a call to null? And I changed the list slice operation to a call to cdr. Then there is some syntactic shuffling, and boom we have scheme code. Normally translating between the two languages won't be this easy, and usually you shouldn't have to, but it's a useful technique to try if you are stuck.

Let's walk through a harder problem now

1.3 Count Sublist Palindromes

Some times problems are hard in ANY language. And this is because sometimes we just have to be creative to come up with the solution. So usually when attacking these problems, it's nice to use a notation called pseudo-code. This isn't very formal or anything, it's code that basically looks like python, but when something is tedious or hard to actually implement, but conceptually simple we can assume that we have a function written that does that . It makes it easier to focus on the crux of the problem.

So lets attack such a problem. Lets say we have a list of integers, we can say that the list is a Palindrome list if the reverse of the list is equal to the list itself. Now our slightly more difficult problem is to find all possible sublists of a list that are palindromes. A sublist (in python-speak) is any slice of the list with step one (that is no third argument to slice). Any list of size 1 is considered a palindrome. List of size 0 are NOT palindromes. Lets do this in python first.

```
def count_palindromes(lst):
        ''''
        >>> count_palindromes([5])
        1
        >>> count_palindromes([5, 7])
        2
        >>> count_palindromes([5, 7, 5])
        4
        # 6 length-1 + 4 length-3 + 2 length-5
        >>> count_palindromes([5, 7, 5, 7, 5, 7])
        12
        # 5 length-1 + 4 length-2 + 3 length-3 + 2 length-4 + 1 length-5
        >>> count_palindromes([5, 5, 5, 5, 5])
        15
        ''''
```

Mull about how you would attack this problem in python with the person sitting next to you. Hint: think recursively.

Ok we know palindromes are by definition recursive (if a list is a palindrome list[1:-1] is also a palindrome). So lets use that to our advantage. If our function is recursive we surely need a base case. More specifically we need to know when we win and when to give up, so we will have a base case where we've successfully found a palindrome and a base case where we failed at finding a palindrome.

1. What is our success base case?

   Solution: When lst is of length 1

2. What is our failure base case?

   Solution: When lst is of length 0

Ok now we have our base cases. Our function looks like

```
def count_palindromes(lst):
        if len(lst) == 0:
                return 0
        if len(lst) == 1:
                return 1
        #RECURSIVE STUFF HERE
```

Now lets think about our recursive cases. How do we represent our problem as a sum of some recursive calls (which are smaller problems)?

If the entire list is a palindrome, we know that our answer is at least one so we can add one to our recursive calls.

1. How do we check if the entire list is a palindrome?

> **Solution:** lst[::-1] == lst

Now our function looks like:

```
def count_palindromes(lst):
        if len(lst) == 0:
                return 0
        if len(lst) == 1:
                return 1
        int(lst[::-1] == lst) + #SOME RECURSIVE STUFF
```

1. We have arrived at the crux of our problem, how do we represent count palindromes(lst) as multiple calls to count palindromes with smaller lists as inputs?

> **Solution:** palindrome_sublists(lst[1:]) + palindrome_sublists(lst[:-1]) - palindrome_sublists(lst[1:-1])

This is kind of confusing lets walk through it slowly. If I want to count ALL the palindromes in all sublists, I'd need to look at some smaller lists. If I count all the palindromes in the sublist lst[1:] (I know this works because I'm taking the recursive leap of faith), I have counted all the possible palindromes that don't include the first element. But thats not okay! So I count all palindromes in sublist lst[:-1], now this has all the palindromes that don't include the last element. So we've covered all the cases right? Wrong! We've double counted all the palindromes in the lst[1:-1] (the middle part of the list). So we need to subtract that part out of our answer.

So gluing all our code together our final solution looks like:

```
def count_palindromes(lst):
        if len(lst) == 0:
                return 0
        if len(lst) == 1:
                return 1
        int(lst[::-1] == lst) + palindrome_sublists(lst[1:]) +
         palindrome_sublists(lst[:-1]) - palindrome_sublists(lst[1:-1])
```

Give yourself, and your neighbors a high five, you've solved a pretty difficult problem!

1.4 Counting Palindromes in Scheme

1. All right now lets do the same problem in Scheme. You may be sad because Scheme doesn't have list slicing, so for the purposes of this problem lets assume you have a lst_slice method that takes three arguments ,a scheme list, and two indices and returns that slice of the list (just like in python). If you don't know how something works in Scheme, just guess! We can iron out the syntax later, lets just get our problem in Scheme land.

   Solution:

```
(define (is_palindrome lst)
  (if (equal? lst (reverse lst))
    1 0)
)

(define (palindrome_sublists lst)
  (if (eq? (length lst ) 0)
    0
  (if (eq? (length lst) 1)
    1
  (let ((l (- (length lst) 1 )))
  (+ (is_palindrome lst)
     (palindrome_sublists (lst_slice lst 1 (+ l 1)))
        (palindrome_sublists (lst_slice lst 0 l)) (- 0
           (palindrome_sublists (lst_slice lst 1 l)))))))))
```

2. Now write lst_slice so the above function can run.

Solution:

```
(define (lst_slice lst i k)
  (if (null? lst) nil
  (if (= i k)
    nil
  (if (> i 0)
    (lst_slice (cdr lst) (- i 1) (- k 1))
    (cons (car lst) (lst_slice (cdr lst) i (- k 1))
  ))
  )))
```