CSSE 304 Assignment 8

Objectives To **Follow the Grammar** and short-circuit when a particular item is found.
To understand how to create "objects" with persistent "fields",

Same rules as the previous assignments. In particular, mutation is not allowed, except for problem 1, whose definition requires a mutation mechanism.
No argument error-checking is required. You may assume that all arguments have the correct form.

You may find *Chez* Scheme's `trace-let`, `trace-lambda`, and `trace-define` to be helpful for debugging your code; if you use them, you should remove them before submitting to the grading server.

#1 (40 points). s-lists are defined on page 8 of EoPL, and we used them in coding examples days 9 and 10. You are to write a procedure called `make-slist-leaf-iterator`. This procedure takes an `s-list` as its argument, and returns an iterator "object" procedure that only has one method, **`'next`**. Each time **`next`** is called, it returns the next symbol from the `s-list`. If the iterator is called again after all of the symbols from the `s-list` have been returned, it returns `#f`. An example should help you to understand what an s-list leaf iterator is supposed to do (things in **bold** are the things that I typed, the others are Scheme's responses):

```
> (define iter (make-slist-leaf-iterator '((a (b c) () d) () e)))
> (iter 'next)
a
> (iter 'next)
b
> (iter 'next)
c
> (iter 'next)
d
> (iter 'next)
e
> (iter 'next)
#f
> (iter 'next)
#f
```

The iterator procedure must maintain a mutable state if it is to exhibit this behavior. Mutation is allowed for this problem.

One simple approach to creating the iterator would be to simply call `flatten` on the s-list, and then use `cdr` to traverse the resulting flat list. However, this approach has a property that no iterator should have! It requires visiting every symbol in the s-list before the iterator is ever asked to return a symbol. If we create an iterator procedure for an s-list that contains thousands of symbols, but then we call that iterator only a few times, the iterator should not have to deal with all of the symbols in the s-list. Thus you may not use this approach.

Thus, due to efficiency, you are not allowed to use any approach that "preprocesses" the entire s-list in order to make the iteration simpler. It must only visit the elements of the s-list that are necessary to process all of the calls to `next`.

Should an s-list leaf iterator be based on preorder or postorder? It doesn't matter; we are only iterating the leaves; preorder and postorder visit the leaves in the same order. Preorder is probably easier to code.

The standard way to do tree iterators is to use a stack to keep track of subtrees whose left side we have already visited, but have not visited the right side. You must create and use a stack object *via* the `make-stack` procedure from class (code is below). In this case, the stack will keep track of `cdr`s of the pairs whose `car`s we have already visited. The idea is similar to the tree iterators presented in chapter 18 of the CSSE 230 book: Mark Allen Weiss, *Data Structures and Problem Solving using Java*. But the EoPL s-list structure allows the Scheme code a little bit simpler than Weiss's Java preorder iterator.

Be careful about **empty sublists**. Notice in the example above that the iterator skips them.

In my code, an s-list leaf iterator "object" procedure has only one persistent local variable, whose value is a stack object.

Here is my code from class for constructing a stack:

```
(define make-stack
 (lambda ()
  (let ([stk '()])
   (lambda (msg  . args )
    (case msg   ; Scheme's case is a similar to switch in some other languages.
      [(empty?) (null? stk)]
      [(push)   (set! stk (cons (car args) stk))]
      [(pop)    (let ([top (car stk)])
                   (set! stk (cdr stk))
                   top)]
      [else (errorf 'stack "illegal message to stack object: ~a" msg)])))))
```

Listing the code here is a reminder that you make sure that you understand this code and how it makes a persistent local variable, and why the original code from class does not work.

#2 (40 points) On pp 20-22 of EoPL, you should have read about `(subst new old slist)`, which substitutes *new* for each occurrence of symbol *old* in the s-list *slist*. We also wrote this procedure during Session 9 (in Fall, 2015 term; may happen in a different session in a later term).

Now write `subst-leftmost`, which takes the same arguments (plus a comparison predicate, described below), but only substitutes *new* for the **leftmost** occurrence of *old*. By "leftmost", I mean the occurrence that would show up first if Scheme printed the s-list. Another way of saying it is "the one that is encountered first in a preorder traversal". Your procedure must “short-circuit”, i.e. avoid traversing the `cdr` of any sublist if the substitution has already been done anywhere in the `car` of that sublist. You should only traverse the parts of the tree that are necessary to find the leftmost occurrence and do the substitution, then copy the references to all of the remaining `cdr`s without traversing the sublists of those `cdr`s. Also, ***you must not traverse the same sublist twice*.**
Hint: if your code calls `equal?` or `contains?` or any other procedure that traverses an entire s-list, you are probably violating the “don't-traverse-twice” rule.

In order to make the procedure slightly more general (and easier for me to test the above constraint), *subst-leftmost* will have an additional argument that *subst* does not have. It is an equality procedure, used to determine whether an individual symbol or number in the list matches *old*.

subst-leftmost: *Symbol* × *Symbol* × *S-list* × *Predicate* ➔ *S-list*

```
(subst-leftmost 'k 'b '((c d a (e () f b (c b)) (a b)) (b)) eq?) ➔
                  ((c d a (e () f k (c b)) (a b)) (b))

(subst-leftmost 'b 'a '(c (A e) a d)
     (lambda (x y) (string-ci=? (symbol->string x) (symbol->string y)))) ➔ (c (b e) a d)
```

```
(define subst-leftmost   ; substitute new for leftmost occurrence of old in slist (match defined by equality-pred?)
  (Lambda (new old slist equality-pred?)    ; you fill in the rest.
        ... ))
```

Note: Mutation could possibly be used to do in this problem, but I want you to get a bit more practice on purely functional programming, and this problem will certainly give you that practice! It has a lot in common with `bt-max-interior` from a previous assignment. So **no mutation is allowed**.

Notes, Questions and answers from previous terms’ Piazza:

The iterator problem:

Scheme: Returning a value AND mutating the slist.

>(define lst `(a b c d))

How do I return the car of a, but also *set! lst* to the cdr of *lst* all in one procedure call?

the students' answer,
where students collectively construct a single answer
Let blocks can have multiple expressions in them, and the result of the last expression will be the result of the whole block. For example:

```
  (let ((a (car lst)))
    (set! lst (cdr lst))
    a)
```

the instructors' answer,
where instructors collectively construct a single answer
Actions
Cons, lambda, letrec, begin, and the individual cases of cons and case can also have multiple expressions. They get executed in order, and the value of the last expression is returned.

Subst-leftmost problem:

subst-leftmost: the main indicators that you are violating the rules (instructor note)

Summary of the rules:
1. Don't go through any sublist twice
2. If you did a substitution in the car, do not call the recursive procedure on the cdr,

Some things that will make me suspect that you may have violated the rules:
1. You call **equal?**
2. You call **list?**
3. You call **append**, where the first argument can have length that is greater than 2.
4, You call your recursive procedure on the cdr without first checking to see if you made a substitution in the car.

If you do one of these, I will look closely at your code; most likely you will lose about 10 points for list? , 25 points for equal? and 15 for append

Don't forget to **FOLLOW THE GRAMMAR**. For s-lists, that means that there are three main cases: list is empty, car is a symbol, car is a list (this can be the else case). See the slist examples i the live-in-class folder.

A8: - make-slist-leaf-iterator MUST use my stack class

I want you get experience in both creating a new class and using an existing class. So the stack you use must be created by calling (make-stack). You are allowed to add new methods to that class, but I doubt that you will have to so so.

existence of an answer for subst-leftmost

Can we assume that there always will be at least one symbol in the list that we need to substitute? For example, if we need to substitute a with b, will there always be b in the list?

Students' answer:
No. Take a look at the first test case:

```
(subst-leftmost 'k 'b '() eq?)
```

The expected answer is the empty list.

Ideas for using stacks to reach leaves of sublist in A8 Problem 1?

I'm trying to figure out how to use stacks to iterate through the elements of a sublist.

I have written a sublist out as a binary tree where the left child is the car and the right child is the cdr and can see why the leaves of such a tree would return the elements of a sublist in order, but I can't figure out how I'm supposed to use stacks to get to these leaves.

I tried an approach similar to a preorder traversal, but I had trouble figuring out how much I should push / pull to know which combination of cars and cdrs I am in the sublist.

Am I using the right approach for this problem? What should I be doing differently?

Students' answer:

You can start by pushing the entire s-list onto the stack. Each iteration you pop the top item off. If the car is a symbol you push the cdr onto the stack and return the car. Otherwise, you can push the cdr then the car and recurse.

```
                 return a                                      return a         return b         return a
                    ->                     ->    (a (b))     ->       (b)     ->               ->
(a ((a (b)) c))          ((a (b)) c)           (c)                 (a)              (a)
()
```