CS61A Notes 04 – Lists [Solutions v1.0]

Pair Up!

QUESTIONS: What do the following evaluate to?

```
(define u (cons 2 3))  (define w (cons 5 6))  (define x (cons u w))
(define y (cons w x))  (define z (cons 3 y))
```

1. **u, w, x, y, z (write them out in Scheme's notation)**
```
u: (2 . 3)
w: (5 . 6)
x: ((2 . 3) 5 . 6)
y: ((5 . 6) (2 . 3) 5 . 6)
z: (3 (5 . 6) (2 . 3) 5 . 6)
```

   Note: when you type this into STk, y and z look weird; I'm not sure why right now, but I'll look into it. As data structures, they still work fine.

2. **(car y)**
   (5 . 6)

3. **(car (car y))**
   5

4. **(cdr (car (cdr (cdr z))))**
   3

5. **(+ (cdr (car y)) (cdr (car (cdr z))))**
   12

6. **(cons z u)**
   ((3 (5 . 6) (2 . 3) 5 . 6) 2 . 3)

7. **(cons (car (cdr y)) (cons (car (car x)) (car (car (cdr z)))))**
   ((2 . 3) 2 . 5)

Then Came Lists

QUESTIONS:

1. **Define a procedure `list-4` that takes in 4 elements and outputs a list equivalent to one created by calling list.**
```
(define (list-4 e1 e2 e3 e4)
   (cons e1 (cons e2 (cons e3 (cons e4 '())))))
```

2. **Define a procedure `length` that takes in a list and returns the number of elements within the list.**
```
(define (length ls)
      (if (null? ls)
          0
          (+ 1 (length (cdr ls)))))
```

3. **Define a procedure `list?` that takes in something and returns `#t` if it's a list, `#f` otherwise.**

```
(define (list? ls)
      (or (null? ls)
          (and (pair? ls)
               (list? (cdr ls)))))
```

Where's the base case?!

4. **Define `append` for two lists.**

```
(define (append ls1 ls2)
(if (null? ls1)
    ls2
    (cons (car ls1) (append (cdr ls1) ls2))))
```

5. **Suppose we have `x` bound to a mysterious element. All we know is this:**
`(list? x) ==> #t`
`(pair? x) ==> #f`
What is `x`?
The only thing that's a list but not a pair is '(), the null list.

6. **Add in procedure calls to get the desired results. The blanks don't need to have anything:**

```
(cons 'a '(b c d e))
      ==> (a b c d e)

(append '(cs61a is) (list 'cool ))
      ==> (cs61a is cool)

(cons '(back to) '(save the universe))
      ==> ((back to) save the universe)

(cons '(I keep the wolf) (car '((from the door)) ) )
      ==> ((I keep the wolf) from the door)
```

7. **Define a procedure `(insert-after item mark ls)` which inserts `item` after `mark` in `ls`.**

```
(define (insert-after item mark ls)
      (cond ((null? ls) '())
            ((equal? (car ls) mark)
             (cons (car ls) (cons item (cdr ls))))
            (else (cons (car ls) (insert-after item mark (cdr ls))))))
```

(Slightly) Harder Lists

1. **Define a procedure `(depth ls)` that calculates how maximum levels of sublists there are in `ls`. For example,**
`(depth '(1 2 3 4)) ==> 1`
`(depth '(1 2 (3 4) 5)) ==> 2`
`(depth '(1 2 (3 4 5 (6 7) 8) 9 (10 11) 12)) ==> 3`
Remember that there's a procedure called `max` that takes in two numbers and returns the greater of the two.

```
(define (depth ls)
      (if (atom? ls)
          0
          (max (+ 1 (depth (car ls))) (depth (cdr ls)))))
```

```
You probably need a while to convince yourself this is right.  Why add
1 to the depth of car but not to the depth of cdr?
```

2. **Define a procedure `(remove item ls)` that takes in a list and returns a new list with `item` removed from `ls`.**

```
(define (remove item ls)
      (cond ((null? ls) '())
            ((equal? item (car ls)) (remove item (cdr ls)))
            (else (cons (car ls) (remove item (cdr ls))))))
```

3. **Define a procedure `(unique-elements ls)` that takes in a list and returns a new list without duplicates. You've already done this with `remove-dups`, and it used to do this:**
`(remove-dups '(3 5 6 3 3 5 9 8)) ==> (6 3 5 9 8)`
where the *last* occurrence of an element is kept. We'd like to keep the *first* occurrences:
`(unique-elements '(3 5 6 3 3 5 9 8)) ==> (3 5 6 9 8)`
Try doing it without using `member?`. You might want to use `remove` above.

```
(define (unique-elements ls)
   (if (null? ls)
       '()
       (cons (car ls) (unique-elements (remove (car ls) (cdr ls))))))
```

4. **Define a procedure `(count-of item ls)` that returns how many times a given item occurs in a given list; it could also be in a sublist. So,**
`(count-of 'a '(a b c a a (b d a c (a e) a) b (a))) ==> 7`

```
(define (count-of item ls)
   (cond ((null? ls) 0)
         ((pair? (car ls))
          (+ (count-of item (car ls))
             (count-of item (cdr ls))))
         ((equal? item (car ls)) (+ 1 (count-of item (cdr ls))))
         (else (count-of item (cdr ls)))))
```

5. **Define a procedure `(interleave ls1 ls2)` that takes in two lists and returns one list with elements from both lists interleaved. So,**
`(interleave '(a b c d) (1 2 3 4 5 6 7)) ==> (a 1 b 2 c 3 d 4 5 6 7)`

```
(define (interleave ls1 ls2)
   (cond ((null? ls1) ls2)
         ((null? ls2) ls1)
         (else (cons (car ls1) (interleave ls2 (cdr ls1))))))
```

6. **Write a procedure `(apply-procs procs args)` that takes in a list of single-argument procedures and a list of arguments. It then applies each procedure in `procs` to each element in `args` in order. It returns a list of results. For example,**
`(apply-procs (list square double +1) '(1 2 3 4))`
`   ==> (3 9 19 33)`

```
(define (apply-procs procs args)
   (if (null? procs)
       args
       (apply-procs (cdr procs) (map (car procs) args))))
```

Expression Lists

QUESTIONS

1. **Define a procedure `(eval-plus exp)` that takes in a valid Scheme expression consisting only of + and numbers, and evaluates it to the correct value. Assume that + always only gets two arguments. For example,**
   `(eval-plus 3) ==> 3`
   `(eval-plus '(+ 3 4)) ==> 7`
   `(eval-plus '(+ 10 (+ 3 2)) ==> 15`

```
(define (eval-plus exp)
   (cond ((atom? exp) exp)
         (else (+ (eval-plus (cadr exp)) (eval-plus (caddr exp))))))
```

2. **(HARD!) Define `(eval-plus exp)` again, but let + take any number of arguments.**

```
We're going to do "mutual recursion":
(define (eval-plus exp)
   (if (atom? exp)
       exp
       (add-expressions (cdr exp))))

(define (add-expressions exps)
   (if (null? exps)
       0
       (+ (eval-plus (car exps)) (add-expressions (cdr exps)))))

Note that eval-plus calls add-expressions to add up a list of
expressions, and add-expressions calls eval-plus to find out the value
of each expression it is given!
```

3. **We'd like some easy way of creating a `lambda` expression. Write `(make-lambda args body)` that takes in the argument list and the body of a procedure, and produces the corresponding `lambda` expression. For example,**
   `(make-lambda '(x y) '(+ x (* y x))) ==> (lambda (x y) (* y x))`

```
(define (make-lambda args body)
   (list 'lambda args body))
```

4. **Recall that there are two ways of defining procedures: the "real" way, and the sugar-coated way. Write a procedure `(unsugar def)` that takes in a procedure definition in sugar-coated syntax, and returns the same definition without using the syntactic sugar. For example,**
   `(unsugar '(define (square x) (* x x)))`
   `==> (define square (lambda (x) (* x x)))`

```
Let's define a few helpers to help us:
(define (def-name def) (caadr def))
(define (def-args def) (cdadr def))
(define (def-body def) (caddr def))

(define (unsugar def)
```

```
    (list 'define (def-name def)
          (make-lambda (def-args def) (def-body def))))
```

5. **Recall that a `let` expression is actually just a `lambda` expression. Write a procedure `(let->lambda exp)` that takes in a `let` expression and returns the corresponding `lambda` expression. For example,**
   `(let->lambda '(let ((x 3) (y 10)) (+ x y)))`
   `   ==> ((lambda (x y) (+ x y)) 3 10)`

   Again, we'll use some helpers to make our code more readable:

```
(define (let-vars exp) (map car (cadr exp)))
(define (let-vals exp) (map cadr (cadr exp)))
(define (let-body exp) (caddr exp))

(define (let->lambda exp)
   (cons (make-lambda (let-vars exp) (let-body exp)) (let-vals exp)))
```