CS61A Notes – Week 4: Pairs and lists, data abstraction

Pair Up!

QUESTIONS: What do the following evaluate to?

IN CLASS:

```
(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)
```

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
```

EXTRA PRACTICE:

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)
```

Don't You Mean `sentence`?

QUESTIONS:

IN CLASS:

1. **Define a procedure `list-4` that takes in 4 elements and outputs a list equivalent to one created by calling `list` (use `cons`!).**

```
(define (list-4 e1 e2 e3 e4)
   (cons e1 (cons e2 (cons e3 (cons e4 '())))))
```

2. **Write `num-satisfies` that takes in a sentence and a predicate, and returns the number of elements in the sentence that satisfy the given predicate. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.**

```
(define (num-satisfies sent pred) (count (keep pred sent)))
```

3. **Rewrite question 2 to take in a *list* and a predicate. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.**

```
(define (num-satisfies ls pred) (length (filter pred ls)))
```

4. **Suppose we have `x` bound to a mysterious element. All we know is this:**
   `(list? x) => #t`
   `(pair? x) => #f`
   What is `x`?

```
The empty list
```

5. **Add in procedure calls to get the desired results. All the parens below represent lists. Blanks can be left blank:**

```
(    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)
```

EXTRA PRACTICE:

6. **Write `all-satisfies` that takes in a sentence and a predicate, and returns `#t` if and only if all of the elements in the list satisfy the given predicate. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.**

```
(define (all-satisfies sent pred) (empty? (keep (lambda (w) (not (pred w))) sent))
(define (all-satisfies sent pred) (= (count (keep pred sent)) (count sent)))
```

7. **Rewrite question 3 to take in a *list* and a predicate. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.**

```
(define (all-satisfies ls pred) (null? (filter (lambda (w) (not (pred w))) ls))
(define (all-satisfies ls pred) (= (length (filter pred ls)) (length sent)))
```

8. **Write `repeat-evens` that takes in a sentence of numbers and repeats the even elements twice. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.**

```
(define (repeat-evens sent) (every (lambda (n) (if (even? n) (se wd wd) wd)) sent))
```

9. **(Hard!) Rewrite question 8 to work on a list of numbers. USE ONLY HIGHER ORDER FUNCTIONS; DO NOT USE RECURSION.**

```
(define (repeat-evens ls)
   (accumulate append ‘()
               (map (lambda (n) (if (even? n) (list n n) (list n))) ls)))
```

10. **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))))))
```

Box and Pointer Diagrams

QUESTIONS: Evaluate the following, and draw a box-and-pointer diagram for each. (Hint: It may be easier to draw the box-and-pointer diagram first.)

IN CLASS:

1. **`(cons (cons 1 2) (cons 3 4))`**

```
((1 . 2) 3 . 4)
```

2. **`(cons '((1 a) (2 o)) '(3 g))`**

```
(((1 a) (2 o)) 3 g)
```

3. **`(list '((1 a) (2 o)) '(3 g))`**

```
(((1 a) (2 o)) (3 g))
```

4. **`(append '((1 a) (2 o)) '(3 g))`**

```
((1 a) (2 o) 3 g)
```

EXTRA PRACTICE:

5. **`(cdr (car (cdr '(((1) 3) (4 (5 6))) )))`**

```
((5 6))
```

6. **`(map (lambda (fn) (cons fn (fn 6))) (list square 1+ even?))`**

```
((#[square] . 36) (#[1+] . 7) (#[even?] . #t))
```

(Slightly) Harder Lists

QUESTIONS:

IN CLASS:

1. **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))))))
```

2. **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))))))
```

3. **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))))))
```

EXTRA PRACTICE:

4. **Define a procedure `(count-unique ls)` which, given a list of elements, returns a list of pairs whose `car` is an element and whose `cdr` is its number of occurrences in the list. For example,**
   `(count-unique '(a b b b c d d a e e f a a))`
   `      => ((a . 4) (b . 3) (c . 1) (d . 2) (e . 2) (f . 1))`
   You might want to use `unique-elements` and `count-of` defined above.

```
(define (count-unique ls)
   (map (lambda(x) (cons x (count-of x ls))) (unique-elements ls)))
```

5. **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))))
```

Data Abstraction

QUESTIONS:

1. **Write a procedure `get-last-names` that takes a list of student records (that we defined above) and returns a sentence of each student's last name. Respect the data abstraction! (Hint: What's the domain of get-last-names? What's the range?)**

```
(define (get-last-names records)
   (map (lambda (rec) (last-name rec)) records))
```

2. **Let's say we wanted to change our internal representation for names so that last names come first in the pair. Which selectors and constructors do we have to change? Modify the procedures that require changes so that they work with the new internal representation.**

```
(define (make-name first last) (cons last first))
(define (first-name student) (cdr (car student)))
(define (last-name student) (car (car student)))
```

3. **Modify the representation of student records to accommodate a student's GPA. You may use list as well as cons for your new representation. Finally, modify and/or create any new selectors or constructors for your new representation.**

```
The easiest way would be to change the student record into a list of three items:

((brian . harvey) 176 4)

(define (make-stduent name sid gpa) (list name sid gpa)
(define (student-id student) (cadr student))
(define (gpa student) (caddr student))
```