Guerrilla Section 4: Scheme

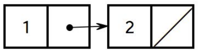
Instructions

Form a group of 3-4. Start on Question 1. Check off with a lab assistant when everyone in your group understands how to solve Question 1. Repeat for Question 2, 3, etc. **You are not allowed to move on from a question until you check off with a lab assistant.** You are allowed to use any and all resources at your disposal, including the interpreter, lecture notes and slides, discussion notes, and labs. You may consult the lab assistants, **but only after you have asked everyone else in your group. The purpose of this section is to have all the students working together to learn the material.**

Question 1: What would Scheme display?

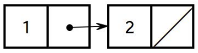
What will Scheme output? Draw the box and pointer whenever the expression evaluates to some pair or list.

```
scm> (cons 1 (cons 2 nil))
```



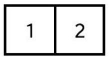
```
(1 2)
```

```
scm> (cons 1 (cons 2 '()))
```



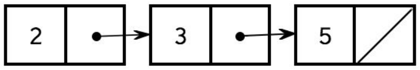
```
(1 2)
```

```
scm> (cons 1 2)
```



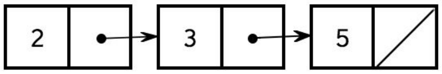
```
(1 . 2)
```

```
scm> '(2 3 5)
```



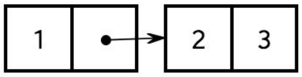
```
(2 3 5)
```

```
scm> '(2 . (3 . (5 . ())))
```



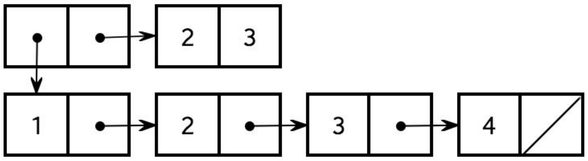
```
(2 3 5)
```

```
scm> (cons 1 (cons 2 3))
```

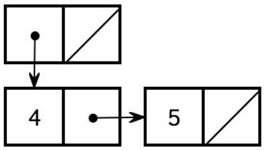


```
(1 2 . 3)
```

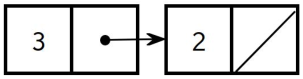
```
scm> (cons (cons (car '(1 2 3))
             (list 2 3 4))
        (cons 2 3))
```



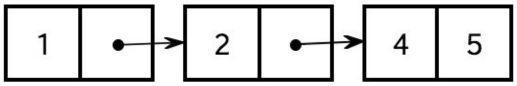
```
 ((1 2 3 4) 2 . 3)
scm> (car (cdr (cdr (car '((1 2 a) a (4 5))))))
 a
scm> (define (cddr x) (cdr (cdr x)))
 cddr
scm> (cddr '((1 2) 3 (4 5)))
```



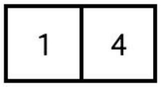
```
 ((4 5))
scm> (define (caar x) (car (car x)))
 caar
scm> (caar '((1 2) 3 (4 5)))
 1
scm> '(((1 . 2) . 3) 4 . (5 . 6))
 (((1 . 2) . 3) 4 5 . 6)
scm> (define a (cons 1 (cons 2 nil)))
 a
scm> a
 (1 2)
scm> (set-car! a 3)
scm> a
```



```
 (3 2)
scm> (set-cdr! (cdr a) (cons 4 5))
scm> a
```



```
 (3 2 4 . 5)
scm> (define lst '(1 2 3))
 lst
scm> (define var 4)
 var
scm> (set-cdr! lst var)
scm> lst
```

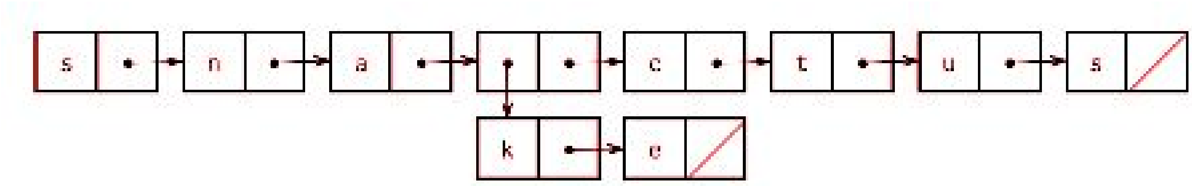


```
 (1 . 4)
```

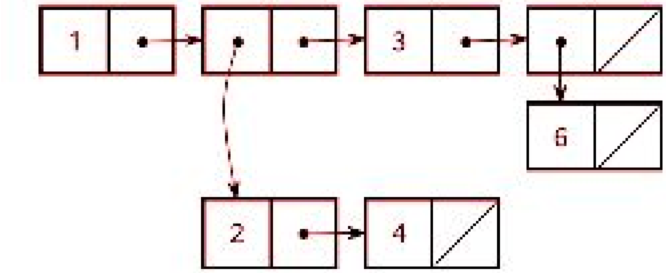
Question 2: Linked List Diagrams

Draw a box-and-pointer diagrams for the following commands:

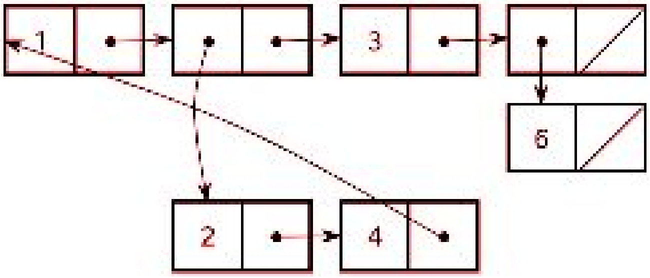
a) `(cons 's (cons 'n (cons 'a (cons (cons 'k (cons 'e nil)) (cons 'c (cons 't (cons 'u (cons 's nil))))))))`



b)
```
scm> (define a '(1 (2 4) 3 (6)))
scm> a
```



c)
```
scm> (set-cdr! (cdr (car (cdr a))) a)
scm> a
```



Question 3: Spot the Bug

```
scm> (sum-every-other '(1 2 3))
4
scm> (sum-every-other '())
0
scm> (sum-every-other '(1 2 3 4))
4
scm> (sum-every-other '(1 2 3 4 5))
9
```

Spot the bug(s), and rewrite the function so it behaves according to the above doctests.

```
(define (sum-every-other lst)
 (cond ((null? lst) lst)
       (else (+ (cdr lst)
                (sum-every-other (caar lst)) ))))
```

1. The base case should return `0`, not `'()`.
2. `(cdr lst)` is a list, so it doesn't make sense to add it to something. Instead, use `(car lst)`, which will give us a number.
3. Using the `caar` (`car` of the `car`) is incorrect because the `car` is a number and it doesn't make sense to get the `car` of a number. Instead, we should use `cddr` (the `cdr` of the `cdr`) to skip forward two elements. However, the `cdr` could be `'()`, so we need to add a case to our `cond` to take care of this.

Here is the corrected function:

```
(define (sum-every-other lst)
  (cond ((null? lst) 0)
        ((null? (cdr lst)) (car lst))
        (else (+ (car lst)
              (sum-every-other (cddr lst)) ))))
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!

Scheme Functions

Question 4: Reverse, HOF Scheme

a) Define `reverse` which takes in a list `lst` and returns a new list with the elements reversed. You may want to use the built-in `append` function.

```
   scm> (define a '(1 2 3))
    a
   scm> a
    (1 2 3)
   scm> (reverse a)
    (3 2 1)
   scm> a
    (1 2 3)
```

```
(define (reverse lst)
  (if (null? lst)
    lst
    (append (reverse (cdr lst)) (list (car lst)))))
```

b) Write a function `list-of-squares` that takes in a Scheme list `lst` and returns a list of the squares of each element in `lst`.

```
scm> (list-of-squares '())
 ()
scm> (list-of-squares '(1 2 3 4 5))
 (1 4 9 16 25)
```

```
(define (list-of-squares lst)
  (define (squarer lst)
    (if (null? lst)
        (quote ())
        (cons  (* (car lst) (car lst))(squarer (cdr lst)))))
  (squarer lst))
```

Question 5: Add To All

The function `add-to-all` should behave like this:

```
> (add-to-all 1 '())
()
> (add-to-all 'foo '((1 2) (3 4) (5 6)))
((foo 1 2) (foo 3 4) (foo 5 6))
```

Define `add-to-all` below. You may not need to use all of the provided lines.

```
(define (add-to-all item lst)


_____________________________________________________________________


_____________________________________________________________________


_____________________________________________________________________


_____________________________________________________________________

)
```

```
(define (add-to-all item lst)
  (if (null? lst)
    lst
    (cons (cons item (car lst))
          (add-to-all item (cdr lst)))))
```

Question 6: Sublists

Define `sublists`, which takes in a lst and returns all possible sublists. Order doesn't matter.
Hint: use `add-to-all`.

```
scm> (sublists '(1 2 3))
 (() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
```

```
(define (sublists lst)
  (if (null? lst) '(())
    (let ((recur (sublists (cdr lst))))
      (append recur
              (add-to-all (car lst) recur)))))
```

Question 7: Sixty-Ones

Define `sixty-ones`. Return the number of times that 1 follows 6 in the list.

```
scm> (sixty-ones '(4 6 1 6 0 1))
1
scm> (sixty-ones '(1 6 1 4 6 1 6 0 1))
2
scm> (sixty-ones '(6 1 6 1 4 6 1 6 0 1))
3
```

```
(define (sixty-ones lst)
  (cond ((or (null? lst) (null? (cdr lst))) 0)
        ((and (= 6 (car lst)) (= 1 (cadr lst)))
          (+ 1 (sixty-ones (cddr lst))))
        (else (sixty-ones (cdr lst)))))
```

STOP!

Don’t proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!

Question 8: Replace X

a) Write a recursive function `replace-x` that takes in a Scheme list `lst` and returns a new list where every instance of `x` is replaced with `y`.

```
scm> (replace-x '() 1 2)
()
scm> (replace-x '(1 2 3) 3 4)
(1 2 4)
scm> (replace-x '(5 7 8 7) 7 10)
(5 10 8 10)
scm> (replace-x '(1 2 3 3 3) 3 5)
(1 2 5 5 5)
```

```
(define (replace-x lst x y)
  (cond ((null? lst) lst)
        ((eq? (car lst) x) (cons y (replace-x (cdr lst) x y)))
        (else (cons (car lst) (replace-x (cdr lst) x y)))))
```

b) EXTRA Challenge Question: Rewrite `replace-x` such that it takes in a Scheme list `lst` and mutates it, replacing each instance of `x` with `y`.

```
(define (replace-x lst x y)
  (cond ((null? lst) lst)
        ((eq? (car lst) x) (begin (set-car! lst y) (set-cdr! lst (replace-x
(cdr lst) x y)) lst))
        (else (begin (set-cdr! lst (replace-x (cdr lst) x y)) lst)))))
```

Question 9: Sequence in List

Fill in the following function, which checks to see if a particular sequence of items, `sub-lst`, can be found in another scheme list, `lst` (the items must be in order, but not necessarily consecutive). You may not need to use all of the lines to write your code.

```
scm> (seq-in-lst '(1 2 3 4) '(1 3))
#t
scm> (seq-in-lst '(1 2 3 4) '(4 3 2 1))
#f
```

```
(define (seq-in-lst lst sub-lst)

     (cond ________________________________________________________


  ________________________________________________________

          ________________________________________________________


  ________________________________________________________))
```

```
(define (seq-in-lst lst sub-lst)
     (cond ((null? sub-lst) #t)
           ((null? lst) #f)
           ((eq? (car lst) (car sub-lst)) (seq-in-lst (cdr lst) (cdr
     sub-lst)))
           (else (seq-in-lst (cdr lst) sub-lst))))
```

STOP!

Make sure everyone in your group has finished and understands all exercises in this section,
and get checked off!

EXTRA Challenge Question 10: No Elevens

The function `no-elevens` should return a list of all distinct length-n lists of 1s and 6s that do not contain 1 after 1.

```
> (no-elevens 2)
((6 6) (6 1) (1 6))
> (no-elevens 3)
((6 6 6) (6 6 1) (6 1 6) (1 6 6) (1 6 1))
> (no-elevens 4)
((6 6 6 6) (6 6 6 1) (6 6 1 6) (6 1 6 6) (6 1 6 1) (1 6 6 6) (1 6 6
1) (1 6 1 6))
```

Define `no-elevens` below. You may not need all of the lines provided below.

```
(define (no-elevens lst)

   (cond ___________________________________________________________

         ___________________________________________________________

         ___________________________________________________________

         ___________________________________________________________

         ___________________________________________________________

    )
)
```

```
(define (no-elevens n)
   (cond ((= 0 n) '(()) )
         ((= 1 n) '((6) (1)) )
         (else (append (add-to-all 6 (no-elevens (- n 1)))
                       (add-to-all 1
                         (add-to-all 6 (no-elevens (- n 2))))))))
```