Scheme

Instructions

Form a group of 3-4. Start on Question 0. Check off with a lab assistant when everyone in your group understands how to solve Question 0. Repeat for Question 1, 2, etc. **You're 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.**

Scheme

Question 0

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

```
> (or 'false (/ 1 0) 'true)

> '(1 2 3)

> '(1 . (2 . (3 . ())))

> '(((1 . 2) . 3) 4 . (5 . 6))

> (cons 1 2)

> (cons 2 '())


> (cons 1 (cons 2 '()))

> (cons 1 (cons 2 3))

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

> (cadar '((1 2) 3 (4 5)))

> (caddr '((1 2) 3 (4 5)))

> (cddar '((1 2) 3 (4 5)))
> (cddr '((1 2) 3 (4 5)))
```

Question 1

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

Spot the bug(s). Test your answer in the interpreter before talking with a lab assistant/tutor.

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

Question 2

2a. Define `append`. **In Scheme, `append` takes in two lists and makes a larger list.**

```
> (append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
```

2b. Define `reverse`. Hint: use `append`.

```
> (reverse '(1 2 3))
(3 2 1)
```

2c. Define `reverse` without using `append`. Hint: use a helper function and `cons`.

Question 3

3a. Define `add-to-all`.

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

2b. Define map.

```
> (map (lambda (x) (+ x 1)) '(1 2 3))
(2 3 4)
```

3c. Define `add-to-all` using one call to `map`. Hint: this may require a lambda.

Question 4

Define `sublists`. Hint: use `add-to-all`.

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

Question 5

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

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

Question 6

Define `no-elevens`. 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))
```

Exceptions

Question 1
How do we raise exceptions in Python? What type are Exceptions?

Question 2
How do we handle raised exceptions? And why would we need to do so?