CS 61A Spring 2019 | Scheme, Exceptions Guerrilla Section 4: April 6, 2019 Solutions

1 Scheme

Questions

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

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

```
Error
```

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

```
(1 2 3)
```

```
> (cons 2 '())
```

```
(2)
```

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

```
(1 2)
```

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

```
2
```

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

```
(4 5)
```

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

```
()
```

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

```
((4 5))
```

1.2 Spot the bug(s). Test out the code and your fixes in the scheme interpreter! (https://scheme.cs61a.org/)

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

1. Missing a paren at the end.
2. The base case should return 0, not '().
3. (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.
4. 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.

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))))))
```

1.3 Define **append**, which takes in two lists and concatenates them together.

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

```
(define (append lst1 lst2)
  (cond ((null? lst1) lst2)
(else (cons (car lst1) (append (cdr lst1) lst2)))))
```

1.4 Define **reverse**. You may use **append** in your definition.

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

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

1.5 Define **reverse** without using **append**. (Hint: use a helper function and **cons**)

```
(define (reverse lst)
    (define (helper lst reversed)
    (if (null? lst)
      reversed
      (helper (cdr lst) (cons (car lst) reversed ))))
    (helper lst '()))
```

1.6 Define **add-to-all**, which takes in an item and a list of lists, and adds that item to the front of each nested list.

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

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

1.7 Define **map**, which takes in a function and a list, and applies that function to each item in the list.

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

```
(define (map f lst)
  (if (null? lst)
      lst
      (cons (f (car lst)) (map f (cdr lst)))))
```

1.8 Define **add-to-all** using one call to **map**. (Hint: consider using a lambda expression!)

```
(define (add-to-all item lst)
    (map (lambda (inner-lst) (cons item inner-lst)) lst))
```

1.9 Define **sublists**. (Hint: use **add-to-all**)

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

```
Solution 1:
(define (sublists lst)
  (if (null? lst)
    '(())
    (append (sublists (cdr lst))
            (add-to-all (car lst) (sublists (cdr lst))))))

Solution 2: (using let to avoid calling sublists twice)
(define (sublists lst)
  (if (null? lst) '(())
    (let ((recur (sublists (cdr lst))))
      (append recur
              (add-to-all (car lst) recur)))))
```

1.10 Define **sixty-ones**, a funcion that takes in a list and returns 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
```

```
(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)))))
```

1.11 Define **no-elevens**, a function that takes in a number n, and returns a list of all distinct length-n lists of 1s and 6s that do not contain two consecutive 1s.

```
> (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 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))))))))
```

2 Exceptions

Questions

2.1 How do we raise exceptions in Python?

An exception is a object instance with a class that inherits, either directly or indirectly, from the BaseException class. The assert statement introduced in Chapter 1 raises an exception with the class AssertionError. In general, any exception instance can be raised with the raise statement. The general form of raise statements are described in the Python docs. The most common use of raise constructs an exception instance and raises it.

```
>>> raise Exception('An error occurred')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: an error occurred
```

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

An exception can be handled by an enclosing try statement. A try statement consists of multiple clauses; the first begins with try and the rest begin with except:

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
```

The ¡try suite¿ is always executed immediately when the try statement is executed. Suites of the except clauses are only executed when an exception is raised during the course of executing the try suite. Each except clause specifies the particular class of exception to handle.
We want to handle exceptions if we don't want our program to crash immediately when it encounters an error, and if we can anticipate the errors that would occur/have pre-defined ways of handling them.