1 Scheme

1.1 What would Scheme do?

```
scm> (and 0 2 200)
```

```
scm> (or True (/ 1 0))
```

```
scm> (and False (/ 1 0))
```

```
scm> (not 3)
```

1.2 What would Scheme display?

```
scm> (define a (+ 1 2))
```

```
scm> a
```

```
scm> (define b (+ (* 3 3) (* 4 4)))
```

```
scm> (+ a b)
```

```
scm> (= (modulo 10 3) (quotient 5 3))
```

```
scm> (even? (+ (- (* 5 4) 3) 2))
```

```
scm> (if (and #t (/ 1 0)) 1 (/ 1 0))
```

```
scm> (if (> (+ 2 3) 5) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
```

```
scm> ((if (< 9 3) + -) 4 100)
```

```
scm> (if 0 #t #f)
```

1.3 Write two Scheme expressions that are equivalent to the following Python statement - one defining a function directly, and the other creating an anonymous lambda that is then bound to the name `cat`:

```
cat = lambda meow, purr: meow + purr
```

1.4 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)) )))
```

1.5 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
```

1.6 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))
```

1.7 Define `remember`, a function that takes in another zero-argument function `f`, and returns another function `g`. When called for the first time, `g` will call `f` and pass on its return value. When called subsequent times, `g` will remember its previous return value and return it directly, without calling `f` again.

(Hint: look up `set!` in the Scheme spec!)

```
(define (remember f)
```

```
)
scm> (define (f) (print "hello!") 5)
scm> (define g (remember f))
scm> (f)
hello!
5
scm> (g)
hello!
5
scm> (g)
5
```

Check your understanding

- How are call expressions (like `(+ 1 2 3)`) evaluated? What about special forms, like `(or #f #t (/ 1 0))`
- What is the purpose of the `quote` special form?

2 Scheme Lists

2.1 What would Scheme display?

```
scm> (cons 10 (cons 11))


scm> (car (cons 10 (cons 11 nil)))


scm> (cdr (cons 10 (cons 11 nil)))


scm> (cons 5 '(6 7 8))


scm> (define a 10)
a
scm> (list 8 9 a 11)    ; list procedure evaluates all operands


scm> '(8 9 a 11)        ; quote special form does not evaluate operand


scm> (list? (cons 1 2))


scm> (list? (cons 1 (cons 2 '())))


scm> (define null nil)
scm> (equal? null 'null)


scm> (equal? nil 'null)


scm> (equal? null 'nil)


scm> (equal? nil 'nil)


scm> (equal? 'nil ''nil)


scm> (equal? ''nil ''nil)


scm> (eq? ''nil ''nil)
```

2.2 Draw out a box-and-pointer diagram for the following list:

```
scm> (define nested-lst (list 1 (cons 2 (cons 3 'nil)) '(4 5 6) 7))
nested-lst
```

Then, write out what Scheme would display for the following expressions:

```
scm> (cdr nested-lst)
```

```
scm> (cdr (car (cdr nested-lst)))
```

```
scm> (cons (car nested-list) (car (cdr (cdr nested-list))))
```

2.3 Define `concat`, which takes a list of lists, and constructs a list by concatenating all the elements together into one list. Use your `my-append` function to concatenate two lists.

```
(define (concat lsts)

)
scm> (concat '((1 4 7) '(2 5 8)))
(1 4 7 2 5 8)
scm> (concat '((1 4 7) (2 5 8) (3 6 9)))
(1 4 7 2 5 8 3 6 9)
```

Extra

2.4 Notice that the builtin `append` takes in, not a *list* of lists, but an *arbitrary* number of lists as arguments, which it then concatenates together. Implement `better-append`, which behaves in such a manner, allowing the caller to pass in an arbitrary number of arguments. You may use `concat` from the previous question.

(Hint: look up "variadic functions" in the Scheme spec!)

```
scm> (better-append '(1 2 3))
(1 2 3 2 3 4)
scm> (better-append '(1 2 3) '(2 3 4))
(1 2 3 2 3 4)
scm> (better-append '(1 2 3) '(2 3 4) '(3 4 5))
(1 2 3 2 3 4 3 4 5)
```

Check your understanding

- How can you get the third element of a Scheme list? Draw out a box-and-pointer diagram if you aren't sure.
- What is the difference between `eq?` and `equal?` in the context of Scheme lists? Construct two lists `lst1` and `lst2` such that `(equal? lst1 lst2)` is `#t` but `(eq? lst1 lst2)` is `#f`.

3 Tail Recursion

3.1 For the following procedures, determine whether or not they are tail recursive. If they are not, write why not and rewrite the function to be tail recursive on the right.

```
; Multiplies x by y
(define (mult x y)
  (if (= 0 y)
      0
      (+ x (mult x (- y 1)))))

; Always evaluates to true
; assume n is positive
(define (true1 n)
  (if (= n 0)
      #t
      (and #t (true1 (- n 1)))))

; Always evaluates to true
; assume n is positive
(define (true2 n)
  (if (= n 0)
      #t
      (or (true2 (- n 1)) #f)))

; Returns true if x is in lst
(define (contains lst x)
  (cond
    ((null? lst)           #f)
    ((equal? (car lst) x)   #t)
    ((contains (cdr lst) x) #t)
    (else                  #f)))
```

3.2 Tail recursively implement **sum-satisfied-k** which, given an input list **lst**, a predicate procedure **f** which takes in one argument, and an integer **k**, will return the sum of the first **k** elements that satisfy **f**. If there are not **k** such elements, return 0.

```
; Doctests
scm> (define lst `(1 2 3 4 5 6))
scm> (sum-satisfied-k lst even? 2)  ; 2 + 4
6
scm> (sum-satisfied-k lst (lambda (x) (= 0 (modulo x 3))) 10)
0
scm> (sum-satisfied-k lst (lambda (x) #t) 0)
0

(define (sum-satisfied-k lst f k)

)
```

3.3 Tail-recursively implement **remove-range** which, given one input list **lst**, and two nonnegative integers **i** and **j**, returns a new list containing the elements of **lst** except the ones from index **i** to index **j**. You may assume **j** > **i**, and **j** is less than the length of the list. (Hint: you may want to use the built-in **append** function)

```
; Doctests
scm> (append '(1 2) '(3 4) '(5 6))
(1 2 3 4 5 6)
scm> (remove-range '(0 1 2 3 4) 1 3)
(0 4)

(define (remove-range lst i j)
```

```
)
```

Check your understanding

- Why aren't all subexpression evaluations tail-recursive? For instance, why isn't the evaluation of `(+ 4 5)` as part of evaluating `(+ 1 (+ 2 3) (+ 4 5))` tail recursive, even though it's the last expression in the summation?
- Given a function `(f lst)` that acts over a list that has a single recursive call of the form `(f (cdr lst))`, what would be a general approach for rewriting it tail-recursively?

4 Interpreters

4.1 Determine the number of calls to **scheme_eval** and the number of calls to **scheme_apply** for the following expressions. Use the visualizer at `code.cs61a.org` if you're not sure how an expression is evaluated.

```
> (+ 1 2)
3
```

```
> (if 1 (+ 2 3) (/ 1 0))
5
```

```
> (or #f (and (+ 1 2) 'apple) (- 5 2))
apple
```

```
> (define (add x y) (+ x y))
add
> (add (- 5 3) (or 0 2))
2
```

Check your understanding

- When a Scheme interpreter evaluates a combination of the form `(a b c d e)`, when does it evaluate `a`? Does it do so when `a` evaluates to a user-defined function? What about a builtin procedure? What if it is a keyword for a special form?
- What happens when we redefine a builtin procedure, like `#[+]`? For instance, if we run `(define + -)`, and then `(+ 1 2)`, what do we get? What about if we overwrite a keyword corresponding to a special form?