1 Scheme

1.1 What would Scheme do?

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

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

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

```scheme
scm> (not 3)
False
```

1.2 What would Scheme display?

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

```scheme
scm> a
3
```

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

```scheme
scm> (+ a b)
28
```
Scheme & Interpreters

```scheme
scm> (= (modulo 10 3) (quotient 5 3))
#t

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

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

scm> (if (> (+ 2 3) 5) (+ 1 2 3 4) (+ 3 4 (* 3 2)))
13

scm> ((if (< 9 3) + -) 4 100)
-96

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

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

(\(define\) cat (\textbf{lambda} meow purr) (+ meow purr))
(\(define\) (cat meow purr) (+ meow purr))

1.4 Spot the bug(s). Test out the code and your fixes in the scheme interpreter!

(\(define\) (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))))))

1.5 Define sixty-ones, a function 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)))))

1.6 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))

(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))))))))

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.
Scheme & Interpreters

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

(define (remember f)

  (define remembered? #f)
  (define remembered nil)
  (lambda ()
    (if remembered?
        remembered
        (begin (set! remembered (f))
               (set! remembered? #t)
               remembered)))

)

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))

To evaluate call expressions, Scheme first evaluates the operator, and then evaluates all of the operands from left to right. It then applies the operator to the operands (i.e. calls the procedure with the evaluate operands), just like how Python evaluates function calls. In contrast, the first subexpression in a special form is not evaluated, but rather detected and treated specially by the interpreter. The remaining subexpressions may or may not be evaluated, depending on the behavior of the special form. For instance, or will short-circuit when it detects a non-false value, so the above example will not error, since or will never reach the divide-by-zero.

• What is the purpose of the quote special form?

The quote special form is meant to postpone the evaluation of an expression. For instance, if we write (1 2 3), Scheme will typically treat it as a call expression, treating 1 as a procedure (which it is not!). Instead, writing (quote (1 2 3)), or the equivalent shorthand '(1 2 3), will cause the overall expression to evaluate to the second subexpression of the quote special form, allowing us to obtain (1 2 3) after evaluation, as desired.
2 Scheme Lists

2.1 What would Scheme display?

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

Error

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

10

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

(11)

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

(5 6 7 8)

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

(8 9 10 11)

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

(8 9 a 11)

scm> (list? (cons 1 2))

#f

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

#t

scm> (define null nil)
scm> (equal? null 'null)
```
Scheme & Interpreters

#f

scm> (equal? nil 'null)

#f

scm> (equal? null 'nil)

#t

scm> (equal? nil 'nil)

#t

scm> (equal? 'nil ''nil)

#f

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

#t

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

#f

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)
((2 3) (4 5 6) 7)

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

(3)

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

(1 4 5 6)

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.

```scheme
(define (concat lsts)
  (if (null? lsts)
      nil
      (my-append (car lsts) (concat (cdr 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 built-in `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!)

```scheme
(define (better-append . args)
  (concat args))
```

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.

To get the third element of a Scheme list, we need to get the car of the cdr of the cdr of the list - in other words, the third element of lst is (car (cdr (cdr lst))).

• 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.

equal? tests equality, and behaves like == in Python - in other words, it returns true if all the corresponding elements of two lists are themselves equal. eq?, in contrast, tests identity, and returns true only if its two arguments are in fact the same object. Thus, one possibility is simply (define lst1 (list 1)) and (define lst2 (list 1)).
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))))
)

(define (mult x y)
  (define (helper x y total)
    (if (= 0 y)
        total
        (helper x (- y 1) (+ total x))))
  (helper x y 0))

Not tail recursive: after evaluating the recursive call, we still need to apply ‘+’, so evaluating the recursive call is not the last thing we do in the frame.

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

Tail recursive: the recursive call to “true1” is the final sub-expression of the ‘and’ special form. Therefore, we will not need to perform any additional work after getting the result of the recursive call.

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

(define (true2 n)
  (if (= n 0)
      #t
      (true2 (- n 1)))
)

Not tail recursive: the recursive call to “true2” is not the final sub-expression of the ‘or’ special form. Even though it will always evaluate to ‘true’ and short-circuit, the interpreter does not take that into account when determining whether to evaluate it in a tail context or not.
; 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)))

(define (contains lst x)
  (cond
   ((null? lst) #f)
   ((equal? (car lst) x) #t)
   (else (contains (cdr lst) x))))

Not tail recursive: the recursive call to “contains” is in a predicate sub-expression. That means we will have to evaluate another expression if it evaluates to true, so it is not the final thing we evaluate.
3.2 Tail recursively implement \texttt{sum-satisfied-k} which, given an input list \texttt{lst}, a predicate procedure \texttt{f} which takes in one argument, and an integer \texttt{k}, will return the sum of the first \texttt{k} elements that satisfy \texttt{f}. If there are not \texttt{k} such elements, return 0.

; Doctests
 scm> (define \texttt{lst} '(1 2 3 4 5 6))
 scm> (sum-satisfied-k \texttt{lst} even? 2) ; 2 + 4
 6
 scm> (sum-satisfied-k \texttt{lst} (lambda (x) (= 0 (modulo x 3))) 10)
 0
 scm> (sum-satisfied-k \texttt{lst} (lambda (x) #t) 0)
 0

(define (sum-satisfied-k \texttt{lst} \texttt{f} \texttt{k})
  (define (sum-helper \texttt{lst} \texttt{k} \texttt{total})
    (cond
     ((= 0 \texttt{k})
      \texttt{total})
     ((null? \texttt{lst})
      0)
     ((\texttt{f} (car \texttt{lst}))
      (sum-helper (cdr \texttt{lst}) (- \texttt{k} 1) (+ \texttt{total} (car \texttt{lst}))))
     (else
      (sum-helper (cdr \texttt{lst}) \texttt{k} \texttt{total}))))
  (sum-helper \texttt{lst} \texttt{k} 0))

3.3 Tail-recursively implement \texttt{remove-range} which, given one input list \texttt{lst}, and two nonnegative integers \texttt{i} and \texttt{j}, returns a new list containing the elements of \texttt{lst} except the ones from index \texttt{i} to index \texttt{j}. You may assume \texttt{j} > \texttt{i}, and \texttt{j} is less than the length of the list. (Hint: you may want to use the built-in \texttt{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 \texttt{lst} \texttt{i} \texttt{j})
  (define (remove-tail \texttt{lst} \texttt{index} \texttt{so-far})
    (cond
     (> \texttt{index} \texttt{j})
     (append \texttt{so-far} \texttt{lst}))
    (else
     (remove-tail (cdr \texttt{lst}) \texttt{index} (+ \texttt{so-far} (car \texttt{lst})))))
  (remove-tail \texttt{lst} \texttt{i} 0))
(((>= index i)
  (remove-tail (cdr lst) (+ index 1) so-far))
(else
  (remove-tail (cdr lst)
    (+ index 1)
    (append so-far (list (car lst)))))))
(remove-tail lst 0 nil))
)

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?

In most cases, after evaluating a subexpression, we must take what it evaluates to and use it in its enclosing expression. For instance, when evaluating (+ 1 (+ 2 3) (+ 4 5)), after recursively evaluating (+ 4 5) to be 9, we must go back and add it to the other expressions in the summation, before returning 15. However, in tail recursion, we are able to state before evaluating the subexpression that we will just pass its evaluated value back as the evaluated value of the base expression. For instance, when evaluating (and #t (f x)), we know that whatever (f x) evaluates to, that will be what the and expression evaluates to as a whole. Thus, our call to f is in a tail context.

• 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?

There are a number of reasonable strategies - one useful technique is to define a helper function with signature (helper curr rest), where rest stores the portion of the list to be iterated over, and curr contains a single quantity representing the accumulated result over the visited portion of the list. When rest becomes nil, then the helper function can return curr. In the body of the helper function, it can make a single tail call of the form (helper new-curr (cdr rest)), so it iterates over the list in a tail-recursive manner.
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

4 calls to eval:
1. Evaluate entire expression
2. Evaluate +
3. Evaluate 1
4. Evaluate 2
1 call to apply:
1. Apply + to 1 and 2

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

5

6 calls to eval:
1. Evaluate entire expression
2. Evaluate predicate 1
3. Since 1 is true, evaluate the entire sub-expression (+ 2 3)
4. Evaluate +
5. Evaluate 2
6. Evaluate 3
1 call to apply:
1. Apply + to 2 and 3

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

apple

8 calls to eval:
1. Evaluate entire expression
2. Evaluate false
3. Evaluate entire sub-expression (and (+ 1 2) 'apple)
4. Evaluate entire sub-expression (+ 1 2)
5. Evaluate +
6. Evaluate 1
7. Evaluate 2
8. Evaluate 'apple
Since the and expression evaluates to true, we short circuit here.
1 call to apply:
1. Apply + to 2 and 3

>
\[
> \text{(define add x y) (+ x y)}
\]
add
\[
> \text{(add (- 5 3) (or 0 2))}
\]
2

13 calls to eval:
1. Evaluate entire define expression
2. Evaluate entire (add ...) expression
3. Evaluate add operator
4. Evaluate entire (- 5 3) sub-expression
5. Evaluate -
6. Evaluate 5
7. Evaluate 3
8. Evaluate entire (or 0 2) sub-expression
9. Evaluate 0 (and short circuit, since 0 is truthy in Scheme)
10. Evaluate (+ x y) (after applying add and entering the body of the add function)
11. Evaluate +
12. Evaluate x
13. Evaluate y

2 call to apply:
1. Apply - to 5 and 3
2. Apply + to -2 and 0

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?

  If \(a\) evaluates to a user-defined function or a builtin procedure, it will be evaluated. For instance, when evaluating \((+ 1 2 3)\), there are 5 calls to \text{scheme_eval} - the overall expression, the three integers, and the + itself. However, when evaluating a special form, the keyword will not be evaluated, since it is treated specially by the interpreter. For instance, when evaluating \((\text{and} 1 2 3)\), there will only be 4 calls to \text{scheme_eval}, since the symbol \text{and} is a keyword and so will not be evaluated.

- What happens when we redefine a builtin procedure, like \#\([+]\)? For instance, if we run \(\text{(define + -)}\), and then \((+ 1 2)\), what do we get? What about if we overwrite a keyword corresponding to a special form?

  When we redefine a builtin procedure, we will change its behavior, since it is evaluated just like a user-defined procedure. In the above example, after redefining +, \((+ 1 2)\) will evaluate to \(-1\)! However, this is not the case for special forms, since their keywords are checked by the interpreter, not looked up in an environment. Thus, even if we run \(\text{(define and or)}\), \((\text{and} #f #t)\) will still evaluate to \#f, since and will never be evaluated and looked up in the
current environment.