CS 61A
Fall 2019

Scheme

Guerrilla Section : November 1, 2019 Solutions

1 Scheme

1.1 What would Scheme do?

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

```
200
```

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

```
True
```

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

```
False
```

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

```
False
```

1.2 What would Scheme display?

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

```
a
```

```
scm> a
```

```
3
```

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

```
b
```

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

```
28
```

```
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 (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! (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))))))
```

1.5 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
```

```
(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.

(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?

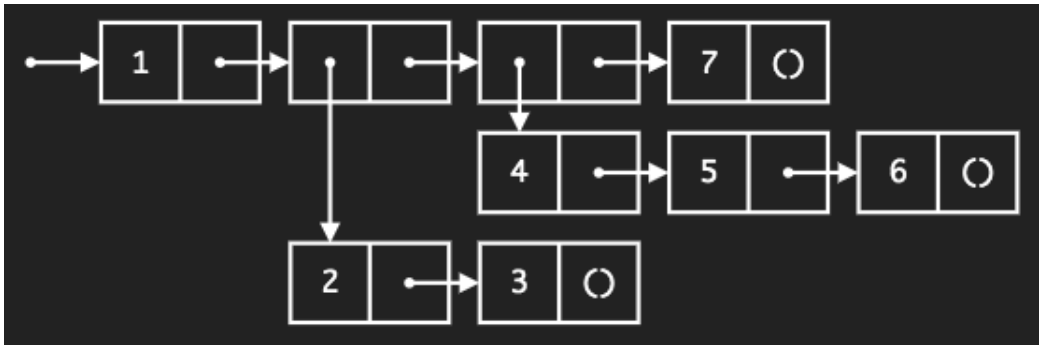
  The `quote` special form is meant to *postpone* the evaluation of an expression. For instance, if we write `(1 2 3)`, Scheme will typically treat it as a call expression, treating `1` as a procedure (which it is not!). Instead, writing

`(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 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)
```

Extra

2.2 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!)

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

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