# CS 61A Fall 2025

# Scheme, Scheme Lists

Discussion 10: November 5, 2025

If there are fewer than 3 people in your group, merge your group with another group in the room. If your group has 6 or more students, you're welcome to split into two sub-groups and then sync up at the end. If you want two separate Pensieve documents for the two sub-groups, just have one sub-group add 1000 to their group number.

Switch to Pensieve:

- **Everyone**: Go to pensieve.co, log in with your @berkeley.edu email, and **enter your group number** as the room number (which was in the email that assigned you to this discussion). As long as you all enter the same number (any number), you'll all be using a shared document.

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Penseive doesn't work, return to this page and continue with the discussion.

## Attendance

Your TA will come around during discussion to check you in. You can start on the worksheet before being checked in; you don't need to wait for your TA to get started.

If you didn't attend for a good reason (such as being sick), fill out this form (within 2 weeks of your discussion): attendance form

## Getting Started

Everybody say your name, and then figure out who most recently pet a dog. (Feel free to share dog photos. Even cat photos are acceptable.)

## Scheme

You can use scheme.cs61a.org to try things out.

An `if` expression has 3 subexpressions:

- The first one, called the *predicate*, is always evaluated to choose between the next two.
- The second one, called the *consequent*, is evaluated if the *predicate*'s value is true.
- The third one, called the *alternative*, is evaluated if the *predicate*'s value is false.

The value of the second or third expression (whichever gets evaluated) is the value of the whole `if` expression.

Only `#f` is a false value in Scheme. All other values are true values, including `#t`.

The logical special forms `and` and `or` can have any number of sub-expressions. Use `and` to check if all its sub-expressions are true values. Use `or` to check if any of them are.

An Example:

```
scm> (define y (+ 1 2))  ; y is now 3
y
scm> (define z (or (> y 4) (> y 5) (< y 5)))  ; z is true because y < 5
z
scm> (+ 7 (if (and (> y 1) z) y 1))  ; the if expression evaluates to 3
10
```

The `expect` form is built into scheme.cs61a.org and checks to see if its first sub expression evaluates to its second one. It is used for testing.

### Q1: Perfect Fit

**Definition**: A perfect square is `k*k` for some integer `k`.

Implement `fit`, which takes non-negative integers `total` and `n`. It returns whether there are `n` **different** positive perfect squares that sum to `total`.

**Important:** Don't use the Scheme interpreter to tell you whether you've implemented it correctly. Discuss! On the final exam, you won't have an interpreter.

```
;;; Return whether there are n perfect squares with no repeats that sum to total
;;;
;;; scm> (fit 10 2)
;;; #t
;;; scm> (fit 9 2)
;;; #f
(define (fit total n)
    (define (f total n k)
        (if (and (= n 0) (= total 0))
            #t
        (if (< total (* k k))
            #f
            'YOUR-CODE-HERE



        )))
    (f total n 1))

(expect (fit 10 2) #t)  ; 1*1 + 3*3
(expect (fit 9 1)  #t)  ; 3*3
(expect (fit 9 2)  #f)  ;
(expect (fit 9 3)  #f)  ; 1*1 + 2*2 + 2*2 doesn't count because of repeated 2*2
(expect (fit 25 1)  #t) ; 5*5
(expect (fit 25 2)  #t) ; 3*3 + 4*4
```

This is a tree recursion problem. Your implementation should try different ways of summing up `n` different perfect squares to see if the sum is `total`. For the example `(fit 10 2)`, the call to `(f 10 2 1)` returns whether there are 2 perfect squares that sum to 10 and are each `1*1` or greater. To figure that out, you'll need to make two recursive calls: - `(f 10 2 2)` which returns whether there are 2 perfect squares that sum to 10 and are each `2*2` or greater

(nope) - `(f 9 1 2)` which returns whether there is 1 perfect square that sums to 9 and is `2*2` or greater (yep: `3*3`)

Once your implementation makes these two recursive calls, it needs to combine them into one result.

Use the `(or _ _)` special form to combine two recursive calls: one that uses `k*k` in the sum and one that does not.

The first should subtract `k*k` from `total` and subtract 1 from `n`; the other should leaves `total` and `n` unchanged.

**Discussion Time:** As a group, come up with one sentence describing how your implementation makes sure that all `n` positive perfect squares are **different** (no repeats). If you're not sure why, talk to a TA!

# Scheme Lists & Quotation

Scheme lists are linked lists. Lightning review:

- `nil` and `()` are the same thing: the empty list.
- `(cons first rest)` constructs a linked list with `first` as its first element. and `rest` as the rest of the list, which should always be a list.
- `(car s)` returns the first element of the list `s`.
- `(cdr s)` returns the rest of the list `s`.
- `(list ...)` takes n arguments and returns a list of length n with those arguments as elements.
- `(append ...)` takes n lists as arguments and returns a list of all of the elements of those lists.
- `(draw s)` draws the linked list structure of a list `s`. It only works on code.cs61a.org/scheme. **Try it now with something like `(draw (cons 1 nil))`.**

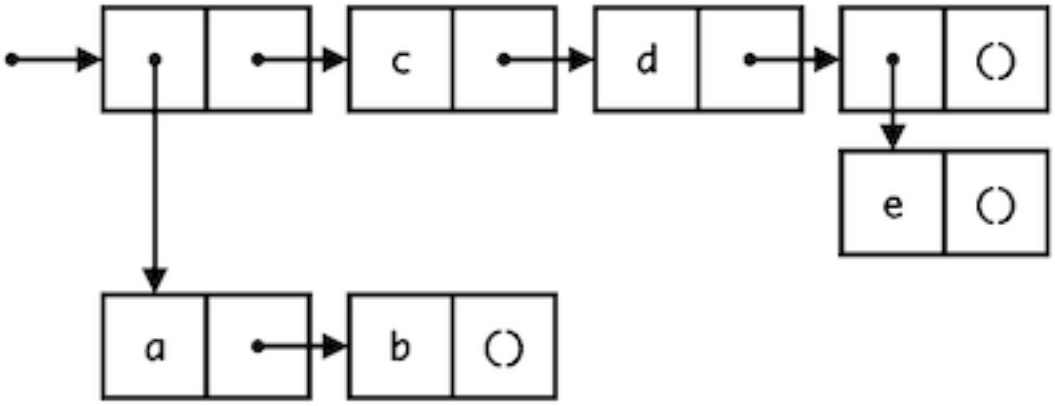
Quoting an expression leaves it unevaluated. Examples: * `'four` and `(quote four)` both evaluate to the symbol `four`. * `'(2 3 4)` and `(quote (2 3 4))` both evaluate to a list containing three elements: 2, 3, and 4. * `'(2 3 four)` and `(quote (2 3 four))` evaluate to a list containing 2, 3, and the symbol `four`.

Here's an important difference between `list` and quotation:

```
scm> (list 2 (+ 3 4))
(2 7)
scm> '(2 (+ 3 4))
(2 (+ 3 4))
```

### Q2: Nested Lists

Create the nested list depicted below three different ways: using `list`, `quote`, and `cons`.




First, describe the list together: "It looks like there are four elements, and the first element is …" If you get stuck, look at the hint below. (But try to describe it yourself first!)

A four-element list in which the first element is a list containing both `a` and `b`, the second element is `c`, the third element is `d`, and the fourth element is a list containing just `e`.

Next, use calls to `list` to construct this list. If you run this code and then `(draw with-list)` in code.cs61a.org, the `draw` procedure will draw what you've built.

```
(define with-list
    (list
        'YOUR-CODE-HERE



    )
)
; (draw with-list)  ; Uncomment this line to draw with-list
```

Every call to list creates a list, and there are three different lists in this diagram: a list containing `a` and `b`: `(list 'a 'b)`, a list containing `e`: `(list 'e)`, and the whole list of four elements: `(list _ 'c 'd _)`. Try to put these expressions together.

Now, use `quote` to construct this list.

```
(define with-quote
    '(
        'YOUR-CODE-HERE



    )

)
; (draw with-quote)  ; Uncomment this line to draw with-quote
```

One quoted expression is enough, but it needs to match the structure of the linked list using Scheme notation. So, your task is to figure out how this list would be displayed in Scheme.

The nested list drawn above is a four-element list with lists as its first and last elements: `((a b) c d (e))`. Quoting that expression will create the list.

Now, use `cons` to construct this list. Don't use `list`. You can use `first` in your answer.

```
(define first
    (cons 'a (cons 'b nil)))
```

```
(define with-cons
    (cons
        'YOUR-CODE-HERE



    )
)
; (draw with-cons)  ; Uncomment this line to draw with-cons
```

The provided `first` is the first element of the result, so the answer takes the form:

```
first ____
```

You can either fill in the blank with a quoted three-element list:

```
'(___ ___ ___)
   c   d  (e)
```

or with nested calls to `cons`:

```
(cons ___ (cons ___ (cons ___ nil)))
       c         d        (e)
```

### Q3: Remove

Implement a procedure `remove` that takes in a list of numbers `s` and a number `x`. It returns a list with all instances of `x` removed from `s`. You may assume that `s` only contains of numbers and will not have nested lists.

```
;;; Return a list with all numbers equal to x removed
;;;
;;; scm> (remove '(3 4 3 4 4 3) 3)
;;; (4 4 4)
;;; scm> (remove '(3 4 3 4 4 3) 4)
;;; (3 3 3)
(define (remove s x)
  (if (null? s) nil
    (if (equal? x ____) ____ ____)
))

(expect (remove '(3 4 3 4 4 3) 3) (4 4 4))
(expect (remove '(3 4 3 4 4 3) 4) (3 3 3))
```

### Q4: Pair Up

Implement `pair-up`, which takes a list `s`. It returns a list of lists that together contain all of the elements of `s` in order. Each list in the result should have 2 elements. The last one can have up to 3.

Look at the examples together to make sure everyone understands what this procedure does.

```
;;; Return a list of pairs containing the elements of s.
;;;
;;; scm> (pair-up '(3 4 5 6 7 8))
;;; ((3 4) (5 6) (7 8))
;;; scm> (pair-up '(3 4 5 6 7 8 9))
;;; ((3 4) (5 6) (7 8 9))
(define (pair-up s)
    (if (<= (length s) 3)
        'YOUR-CODE-HERE




    ))

(expect (pair-up '(3 4 5 6 7 8)) ((3 4) (5 6) (7 8)) )
(expect (pair-up '(3 4 5 6 7 8 9)) ((3 4) (5 6) (7 8 9)) )
```

`pair-up` takes a list (of numbers) and returns a list of lists, so when `(length s)` is less than or equal to 3, return a list containing the list `s`. For example, `(pair-up (list 2 3 4))` should return `((2 3 4))`.

Use `(cons _ (pair-up _))` to create the result, where the first argument to `cons` is a list with two elements: the `(car s)` and the `(car (cdr s))`. The argument to `pair-up` is everything after the first two elements.

**Discussion**: What's the most number of elements `s` can have if `(pair-up (pair-up s))` returns a list with only one element? (Don't just guess and check; discuss!)

7. For example, `(pair-up '(1 2 3 4 5 6 7))` evaluates to `(((1 2) (3 4) (5 6 7)))`, which is a list with one element. In general, `(pair-up s)` can have up to 3 elements. If `s` has 8 or more elements, then `(pair-up s)` will have at least 4 elements, which is too many.

# Optional Question

**Q5: Increasing Rope**

**Definition:** A *rope* in Scheme is a non-empty list containing only numbers except for the last element, which may either be a number or a rope.

Implement `up`, a Scheme procedure that takes a positive integer `n`. It returns a rope containing the digits of `n` that is the shortest rope in which each pair of adjacent numbers in the same list are in increasing order.

**Reminder**: the `quotient` procedure performs floor division, like `//` in Python. The `remainder` procedure is like `%` in Python.

```
(define (up n)
    (define (helper n result)
        (define first (remainder n 10))  ; Using first will shorten your code
        (if (zero? n) result
            (helper
                (quotient n 10)
                'YOUR-CODE-HERE




                )))
    (helper
      (quotient n 10)
      'YOUR-CODE-HERE


    ))

(expect (up 314152667899) (3 (1 4 (1 5 (2 6 (6 7 8 9 (9)))))))
```

Compare `first` to `(car result)` to decide whether to `cons` the value `first` onto the `result` or whether to form a new list that contains `first` and `result` as elements.

To correctly call `helper` from within `up`, build a rope that only contains the last digit of `n`: `(remainder n 10)`.