

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.

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`.

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

```
;;; 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
            (or (f total n (+ k 1)) (f (- total (* k k)) (- n 1) (+ k 1))))))
  (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
```

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:

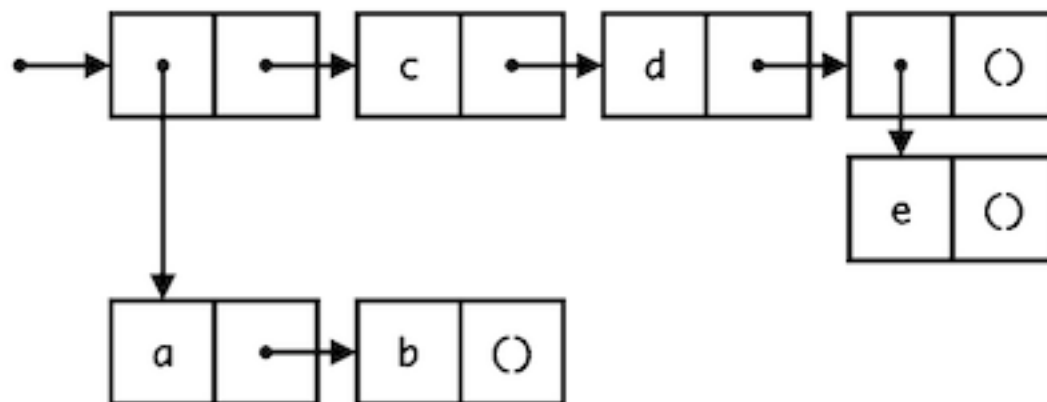
- `'four` and `(quote four)` both evaluate to the symbol `four`.
- `'(2 3 4)` and `(quote (2 3 4))` both evaluate to a list containing three elements: 2, 3, and 4.
- `'(2 3 four)` and `(quote (2 3 four))` evaluate to a list containing 2, 3, and the symbol `four`.

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`.



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
    (list 'a 'b) 'c 'd (list 'e)
  )
)
; (draw with-list) ; Uncomment this line to draw with-list

```

Now, use `quote` to construct this list.

```

(define with-quote
  '(
    (a b) c d (e)
  )
)
; (draw with-quote) ; Uncomment this line to draw with-quote

```

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
    first (cons 'c (cons 'd (cons (cons 'e nil) nil)))
  )
)
; (draw with-cons) ; Uncomment this line to draw with-cons
```

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 (car s)) (remove (cdr s) x) (cons (car s) (remove (cdr s) 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)
      (list s)
      (cons (list (car s) (car (cdr s))) (pair-up (cdr (cdr s))))))

(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)) )
```

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)
         (if (< first (car result))
             (cons first result)
             (list first result))
         )))
  (helper
   (quotient n 10)
   (list (remainder n 10))
  ))

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

Tail Context

When trying to identify whether a given function call within the body of a function is a tail call, we look for whether the call expression is in **tail context**.

Given that each of the following expressions is the last expression in the body of the function, the following expressions are tail contexts:

1. the second or third operand in an **if** expression
2. any of the non-predicate sub-expressions in a **cond** expression (i.e. the second expression of each clause)
3. the last operand in an **and** or an **or** expression
4. the last operand in a **begin** expression's body
5. the last operand in a **let** expression's body

For example, in the expression `(begin (+ 2 3) (- 2 3) (* 2 3))`, `(* 2 3)` is a tail call because it is the last operand expression to be evaluated.

Q6: Is Tail Call

For each of the following procedures, identify whether it contains a recursive call in a tail context. Also indicate if it uses a constant number of active frames.

```
(define (question-a x)
  (if (= x 0) 0
      (+ x (question-a (- x 1)))))
```

In the recursive case, the last expression that is evaluated is a call to `+`. Therefore, the recursive call is not in tail context, and each of the frames remain active. This procedure uses a number of active frames proportional to the input `x`.

```
(define (question-b x y)
  (if (= x 0) y
      (question-b (- x 1) (+ y x))))
```

The recursive call is the third operand in the `if` expression, so it is in tail context. This means that the last expression that will be evaluated in the body of this procedure is the recursive procedure call, so this procedure can be run with a constant number of active frames.

```
(define (question-c x y)
  (if (> x y)
      (question-c (- y 1) x)
      (question-c (+ x 10) y)))
```

The recursive calls are the second and third operands of the `if` expression. Only one of these calls is actually evaluated, and whichever one it is will be the last expression evaluated in the body of the procedure. This procedure therefore can be run with a constant number of active frames.

Note that if you actually try and evaluate this procedure, it will never terminate. But at least it won't crash from hitting max recursion depth!

```
(define (question-d n)
  (if (question-d n)
      (question-d (- n 1))
      (question-d (+ n 10))))
```

The second and third recursive calls are in tail context, but the first is not. Since not all the recursive calls are tail calls, this procedure requires active frames for all of the recursive calls.

Additionally, this question will actually lead to infinite recursion because the if condition will never reach a base case!

```
(define (question-e n)
  (cond ((<= n 1) 1)
        ((question-e (- n 1)) (question-e (- n 2)))
        (else (begin (print 2) (question-e (- n 3))))))
```

The second and third recursive calls are the second expressions in a clause, so they are in tail context. However, the first recursive call is not in tail context. Since not all recursive calls are tail calls, this procedure is not tail recursive and does not use a constant number of active frames.