

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
            '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
```

Q2: Greatest Common Divisor

The Greatest Common Divisor (GCD) is the largest integer that evenly divides two positive integers.

Write the procedure `gcd`, which computes the GCD of numbers `a` and `b` using Euclid's algorithm, which recursively uses the fact that the GCD of two values is either of the following:

- the smaller value if it evenly divides the larger value, or
- the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value

In other words, if `a` is greater than `b` and `a` is not divisible by `b`, then

```
gcd(a, b) = gcd(b, a % b)  # please note that this is Python syntax
```

You may find the provided procedures `min` and `max` helpful. You can also use the built-in `modulo` and `zero?` procedures.

```
>scm> (modulo 10 4)
>2
>scm> (zero? (- 3 3))
>#t
>scm> (zero? 3)
>#f
```

```
(define (max a b) (if (> a b) a b))
(define (min a b) (if (> a b) b a))
(define (gcd a b)
  'YOUR-CODE-HERE

)

(expect (gcd 0 4) 4)
(expect (gcd 10 0) 10)
(expect (gcd 30 5) 5)
(expect (gcd 24 60) 12)
(expect (gcd 1071 462) 21)
```

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))
```

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 to make sure you understand 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)) )
```

Q5: List Insert

Write a Scheme function that, when given an element, a list, and an index, inserts the element into the list at that index. You can assume that the index is in bounds for the list.

```
(define (insert element lst index)
  'YOUR-CODE-HERE

)

(expect (insert 2 '(1 7 9) 2) (1 7 2 9))

(expect (insert 'a '(b c) 0) (a b c))
```

Q6: 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))))))))
```