1. Implement non-contiguous, which takes in two lists, subseq and lst, and returns whether subseq is a non-contiguous subsequence of lst. A sequence is a non-contiguous subsequence if its elements appear in the list in order but not necessarily immediately next to each other.

```
scm> (non-contiguous '() lst)
  True
scm> (non-contiguous '(1 3 6) '(1 2 3 4 5 6))
  True
scm> (non-contiguous '(1 5 2) '(1 2 3 4 5 6))
  False
```

```scheme
(define (non-contiguous subseq lst)
  (cond
    ________________________________
    ________________________________
    ________________________________
    (else ________________________________________________)))
```

Is this procedure properly tail recursive? ____________

Implement assert-equals that takes in an expected value, expected, and an expression, expression, and returns whether the expression evaluates to the expected value to check whether your implementation works!

```
scm> (assert-equals #t '(non-contiguous '(1 3 6) '(1 2 3 4 5 6)))
  ok
scm> (assert-equals #f '(non-contiguous '(1 5 2) '(1 2 3 4 5 6))
  ok
scm> (assert-equals #t '(non-contiguous '(1 5 2) '(1 2 3 4 5 6))
  (expected #t but got #f)
```

```scheme
(define (assert-equals expected expression)
  (if
    ________________________________
    ________________________________
    ________________________________
    (expected #t but got #f))
```

```
2. **Lazy Sunday (Fa14 Final Q4c)** Implement the Scheme procedure `directions`, which takes a number `n` and a symbol `sym` that is bound to a nested list of numbers. It returns a Scheme expression that evaluates to `n` by repeatedly applying `car` and `cdr` to the nested list. Assume that `n` appears exactly once in the nested list bound to `sym`.

*Hint:* The implementation searches for the number `n` in the nested list `s` that is bound to `sym`. The returned expression is built during the search. See the tests at the bottom of the page for usage examples.

```
(define (directions n sym)
  (define (search s exp)
    ; Search an expression `s` for `n` and return an
    ; expression based on `exp`.
    (cond ((number? s) __________________________________)
          ((null? s) nil)
          (else (search-list s exp)))))

(define (search-list s exp)
  ; Search a nested list `s` for `n` and return an
  ; expression based on `exp`.
  (let ((first ________________________________________) (rest _________________________________________))
    (if (null? first) rest first)))
  (search (eval sym) sym))

(define a '(1 (2 3) ((4))))
(define b '(((3 4) 5)))

(directions 1 'a)
; expect (car a)
(directions 2 'a)
; expect (car (car (cdr a)))
(directions 4 'b)
; expect (car (cdr (car b)))
(directions 4 'a)
; expect (car (car (car (cdr (cdr a))))))

What expression will (directions 4 'a) evaluate to? ____________
3. **Interpreters: Implementing Special Forms (Su14 Final Q3)**

In the Scheme project, you implemented several *special forms*, such as if, and, begin, and or. Now we're going to look at a new special form: when. A when expression takes in a *condition* and any number of other subexpressions, like this:

```
(when <condition>
  <exp>
  <exp>
  ...
  <exp>)
```

If *condition* is true, all of the following subexpressions are evaluated in order and the value of the when is the value of the last subexpression. If it is false, none of them are evaluated and the value of the when is okay. For example, `(when (= 1 1) (+ 2 3) (* 1 2))` would first evaluate `(+ 2 3)` and then `(* 1 2).

(a) (2 pt) **Equivalent Scheme Expression** Rewrite the when expression below into another Scheme expression which uses only the special forms you already implemented in your project. That is, create a Scheme expression which does the same thing as this when expression without using when. You should use if in your answer.

```
(when (= days-left 0)
  (print 'im-free)
  'jk-final)
```

You may or may not need to use all of the lines provided.

```
(----------------------------------)
  ----------------------------------
  ----------------------------------

  ----------------------------------)
```

(b) (2 pt) **Box and Pointer** Remember that do when form, like the other do something form functions in the Scheme project, takes in vals and the env to evaluate in. We will be drawing the box-and-pointer diagram for vals above. As an example, the box-and-pointer diagram for `'(+ 1 1)` would be

```
+ 1 1
```

In the example from the description, vals would be `'( (= 1 1) (+ 2 3) (* 1 2))`. Draw the box-and-pointer diagram for this list in the space provided below.
(c) (3 pt) Implementing When Now implement do.when.form. Assume that the other parts of scheme.py have already been modified to accommodate this new special form. You may not need to use all of the lines provided. You do not need to worry about tail recursion. Remember that do.when.form must return two things - a Scheme expression or value and an environment or None.

```python
def do_when_form(vals, env):

```

(d) (2 pt) Implementing Another Special Form Now let’s implement another special form until, which takes in a condition and a series of expressions, and evaluates the expressions in order only if the condition is NOT true. Implement do.until.form using do.when.form. (Remember that Scheme has a built-in not function which your interpreter can evaluate)

```python
def do_until_form(vals, env):

    new_expr = ____________________________________________________________

    return do_when_form(new_expr, env)
```

2 Streams

Repeat n Cycle Implement the Scheme procedure cycle, which takes in a list lst and a positive integer n and returns a Stream containing all the elements in lst repeated n times, with the feature of that the end of the stream points back to the beginning (therefore creating a cycle).

```scheme
scm> (define a (cycle '(1 2 3) 3))

a

scm> (stream-to-list a 20)

(1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2)

(define (cycle lst n)

```

```scheme

```

```scheme
(cond _________________________________________________________________________

_______________________________________________________________________________

_______________________________________________________________________________

s)
```
Stream first n

Implement the Scheme procedure `stream-first-n`, which takes in a positive integer \( n \) and a list \( lst \) and returns a Stream containing all the first \( n \) elements of \( lst \), with the feature of that the end of the stream points back to the beginning.

```
scm> (stream-to-list (stream-first-n 3 '(1 2 3 4)) 10)
(1 2 3 1 2 3 1 2 3 1)
scm> (stream-to-list (stream-first-n 7 '(1 2 3 4)) 10)
(1 2 3 4 1 2 3 4 1 2)
```

```
(define (stream-first-n n lst)
  (define (stream-helper i curr-lst)
    _____________________________________________________________
    __________________________________________________________
    __________________________________________________________
    (stream-helper n lst))

  (stream-helper n lst))
```

3 Macros

Suppose we wish to implement an OOP system in Scheme using macros. We will observe the following restriction: There are only `class`, attributes—no instance attributes. Implement `define-class`, `construct`, `define-method`, `call-method`, and `get-attr` macros below so that the Scheme OOP code has the same effect as the Python OOP code.

===== PYTHON OOP CODE =====
```python
class Dog:
    def age_type():
        if Dog.a < 7:
            return "young"
        elif Dog.a > 7:
            return "old"
        else:
            return "middle aged"

Dog.n = "Fido"
Dog.a = 9
d = Dog()
print(d.age_type())
```
====== SCHEME OOP CODE ======
(define-class Dog)
(define-attr Dog n 'Fido)
(define-attr Dog a 9)
(define-method Dog (age-type)
  (cond
    ((< (get-attr Dog a) 7) 'young)
    ((> (get-attr Dog a) 7) 'old)
    (else 'middle-aged)
  )
)
(define d (construct Dog))
(print (call-method d (age-type)))

====== IMPLEMENTATION ======
(define-macro (define-class class-name)
  `(define ,class-name ____________________________))
(define-macro (construct class-name)
  ________________________________________________)
(define-macro (define-attr class-name attr-name value)
  `(define ,class-name
    (cons
     `(____________________________, ____________________________) ,class-name )))
(define-macro (get-attr class-name attr-name)
  `(begin
    (define (helper class)
      (if (eq? (quote ,attr-name) (car (car class)))
        (car (cdr (car class)))
        (helper (cdr class))))
    (helper ____________________________________________))))
(define-macro (define-method class-name signature body)
  `(define-attr ______________________________))
(define-macro (call-method instance call)
  (cons `(get-attr ,(eval instance) ____________) ____________))