Scheme

**Non-contiguous**

Write a function non-contiguous that checks 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
(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
```

```scm
(define (non-contiguous subseq lst)
  (cond ((null? subseq) #t)
        ((null? lst) #f)
        ((= (car subseq) (car lst)) (non-contiguous (cdr subseq) (cdr lst)))
        (else (non-contiguous subseq (cdr lst))))
```

Is this procedure properly tail recursive? **Yes**

Write a function assert-equals that takes in the expected value of the given expression to check whether your implementation works!

```scm
(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
(define (assert-equals expected expression)
  (if (eq? expected (eval expression))
      'OK
      (list 'expected expected 'but 'got (eval expression))))
```
Lazy Sunday (Fa14 Final Q4c)

(4 pt) 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) (if (= s n) exp nil))
      ((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 (search (car s) (list 'car exp)))
      (rest (search (cdr s) (list 'cdr exp)))
      (if (null? first) rest first)))

    (search (eval sym) sym))

  (define a '(1 (2 3) ((4)))))
  (directions 1 'a)
  ; expect (car a)
  (directions 2 'a)
  ; expect (car (car (cdr a)))
  (define b '((3 4) 5))
  (directions 4 'b)
  ; expect (car (cdr (car b)))

  (d) (2 pt) What expression will (directions 4 'a) evaluate to?

  (car (car (car (cdr (cdr a)))))
Interpreters: Implementing Special Forms (Su14 Final Q12)

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:

\[
\text{(when } \text{<condition>}
\text{ <exp>}
\text{ <exp>}
\text{ ...}
\text{ <exp>)}
\]

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.

\[
\text{(when } (= \text{days-left 0})
\text{ (print 'im-free) 'jk-final)}
\]

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

\[
\text{(if } (= \text{days-left 0})
\text{ (begin (print 'im-free) 'jk-final))}
\]
(b) **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.

```
= -> 1 -> 1
```

(c) **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):
    return Pair("if", Pair(vals.first, Pair(Pair("begin", 
     vals.second), nil))), env
```

(d) **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):
    return do_when_form(Pair(Pair('not', vals.first), 
     vals.second), env)
```
Streams

**Stream-to-List**
Write a function that turns a list into a stream, with the feature of that the end of the stream points back to the beginning (therefore creating a cycle). You must copy the lst `n` times into the stream before creating the 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)
```

```scheme
(scm > (define (cycle lst n)
  (define s (cons-stream (car lst) (helper (cdr lst) n)))
  (define (helper buffer num)
    (cond ((= n 0) s)
      ((null? buffer) (helper lst (- n 1)))
      (else (cons-stream (car buffer) (helper (cdr buffer) n))))))
s)
```

**Cyclic Stream**
Write a function that creates a cyclic stream out of the first `n` elements of `lst`, or the entire `lst` if the length of `lst` is less than `n`.

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

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

```scheme
(define (stream-first-n n lst)
  (define (stream-helper i curr-lst)
    (if (or (zero? i) (null? curr-lst))
      (stream-first-n n lst)
      (cons-stream (car curr-lst) (stream-helper (- i 1) (cdr curr-lst)))))
  (stream-helper n lst))
```
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 the 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 =====
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 nil))
(define-macro (construct class-name)
  `(quote ,class-name))
(define-macro (define-attr class-name attr-name value)
  `(define
    ,class-name
    (cons
      '(,attr-name ,(eval value))
      ,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 ,class-name)
)
(define-macro (define-method class-name signature body)
  `(define-attr ,class-name ,(car signature) (lambda ,(cdr signature)
    ,body))
)(define-macro (call-method instance call)
  (cons `(get-attr ,(eval instance) ,(car call)) (cdr call))
)