FINAL EXAM REVIEW Solutions

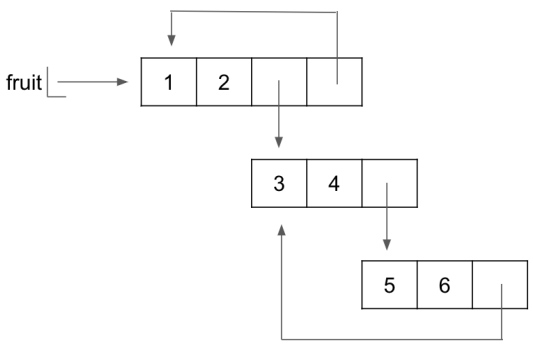
COMPUTER SCIENCE MENTORS 61A

May 6 – May 9, 2025

1 Environment Diagrams

1. Fill in each blank in the code example below so that its environment diagram is the following. You do not need to use all the blanks.

```
fruit = [1, 2, [3, 4]]
fruit._____________________________
fruit[3][2]._____________________
fruit[2][2]._____________________
fruit[3][3][2][2][2][1] = ____
```




```
fruit = [1, 2, [3, 4]]
fruit.append(fruit)
fruit[3][2].append([5, 6])
fruit[2][2].append(fruit[2])
fruit[3][3][2][2][2][1] = 4
```

2 Iterators

2. Define a **non-decreasing path** as a path from the root where each node's label is greater than or equal to the previous node along the path. A **subpath** is a path between nodes X and Y, where Y must be a descendent of X (ex: Y is a branch of a branch of X).

Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback

(a) Write a generator function `root_to_leaf` that takes in a tree `t` and yields all non-decreasing paths from the root to a leaf node, in any order. Assume that `t` has at least one node.

```
def root_to_leaf(t):
    """
    >>> t1 = Tree(3, [Tree(5), Tree(4)])
    >>> list(root_to_leaf(t1))
    [[3, 5], [3, 4]]
    >>> t2 = Tree(5, [Tree(2, [Tree(7), Tree(8)]), Tree(5, [Tree(6)])])
    [[5, 5, 6]]
    """

    if _____________________:

        ___________________________

    for ___________________________:

        if ___________________________:

            for ___________________________:

                ___________________________________
```

```
def root_to_leaf(t):
    if t.is_leaf():
        yield [t.label]
    for b in t.branches:
        if t.label <= b.label:
            for path in root_to_leaf(b):
                yield [t.label] + path
```

The easiest way to approach this is to notice the two blocks of code that are provided: first an if statement, probably referring to a base case, and a for loop, which will probably be the recursive case. From the doctests, we can see that giving the function a tree that just has one node, or in other words `is_leaf()`, returns a list containing just that node.

In our recursive case we want to do two things. First, we want to check if the next branch value really is non-decreasing. Then, if it is, we want to append the result of calling `root_to_leaf` on the branch to the value of our current tree to create a complete path. So we recurse through each of the branches in `t` (`for b in t.branches`), then check if it is nondecreasing (`t.label <= b.label`), then yield our tree's label appended to the recursive call (the last two lines).

(b) Write a generator function `subpaths` that takes in a tree `t` and yields all non-decreasing subpaths that end with a leaf node, in any order. You may use the `root_to_leaf` function above, and assume again that `t` has at least one node.

```
def subpaths(t):

    yield from _____________________________

    for b in t.branches:

        ___________________________________
```

```
def subpaths(t):
    yield from root_to_leaf(t)
    for b in t.branches:
        yield from subpaths(b)
```

We can split this problem into two steps – yielding all subpaths for the current tree that we have, then yielding all subpaths for all other trees within this tree. It is important to realize that each node in the tree is merely a subtree of the original tree to solve this problem.

To yield all non-decreasing subpaths for our current tree (that is all non-decreasing subpaths that start at our current node and end at the leaf nodes), we can just yield from our previous function, `root_to_leaf`, called on that node. For the rest of the subpaths, we want to recursively call `subpaths` on all our child nodes. This will give us all paths that end on the leaf nodes (because `root_to_leaf` ends on the leaf nodes) that start from any child on this tree. It is important to realize that the base case in this situation is implicit. If a leaf node is passed in and reaches the for loop, the for loop finds no items in `t.branches`, and will just terminate without calling the clause inside.

3. In the following problem, we will represent a bookshelf object using dictionaries.

   In the first section, we will set up the format. Here, we will directly work with the internals of the Bookshelf, so don't worry about abstraction barriers for now. Fill in the following functions based on their descriptions (the constructor is given to you):

```
def Bookshelf(capacity):
    """ Creates an empty bookshelf with a certain max capacity. """
    return {'size': capacity, 'books': {}}

def add_book(bookshelf, author, title):
    """
    Adds a book to the bookshelf. If the bookshelf is full,
    print "Bookshelf is full!" and do not add the book.
    >>> books = Bookshelf(2)
    >>> add_book(books, 'Jane Austen', 'Pride and Prejudice')
    >>> add_book(books, 'Daniel Kleppner', 'An Introduction to Mechanics
       5th Edition')
    >>> add_book(books, 'Kurt Vonnegut', 'Galapagos')
    Bookshelf is full!
    """
    if ___________________________________:
        print('Bookshelf is full!')
    else:
        if author in bookshelf['books']:
            ___________________________________
        else:
            ___________________________________


    if len(bookshelf['books']) == bookshelf['size']:
        print('Bookshelf is full!')
    else:
        if author in bookshelf['books']:
            bookshelf['books'][author].append(title)
        else:
            bookshelf['books'][author] = [title]
```

```
def get_all_authors(bookshelf):
    """
    Returns a list of all authors who have at least one book in the
       bookshelf.
    >>> books = Bookshelf(10)
    >>> add_book(books, 'Jane Austen', 'Pride and Prejudice')
    >>> add_book(books, 'Sheldon Axler', 'Linear Algebra Done Right')
    >>> add_book(books, 'Kurt Vonnegut', 'Galapagos')
    >>> get_all_authors(books)
    ['Jane Austen', 'Sheldon Axler', 'Kurt Vonnegut']
    """
    return ____________________________________


        return list(bookshelf['books'].keys())
```

Now, complete the function `most_popular_author` **without breaking the abstraction barrier**. In other words, you are not allowed to assume anything about the implementation of a Bookshelf object, or use the fact that it is a dictionary. You can only use the methods above and their stated return values.

```
def most_popular_author(bookshelf):
    """
    Returns the author with the greatest number of books on this bookshelf.
    You can assume that the bookshelf is not empty.
    >>> books = Bookshelf(100)
    >>> add_book(books, 'Orson Scott Card', 'Xenocide')
    >>> add_book(books, 'Orson Scott Card', 'Children of the Mind')
    >>> add_book(books, 'J.R.R. Tolkien', 'The Hobbit')
    >>> most_popular_author(bookshelf)
    'Orson Scott Card'
    """
    return max(_____________________________________________________,

        key=_____________________________________________________________)


        return max(get_all_authors(bookshelf), key=lambda x:
           len(get_author_books(x))
```

4. Find the $\Theta(\cdot)$ runtime bound for `hiya(n)`. Remember that Python strings are immutable: when we add two strings together, we need to make a copy.

```
def hiii(m):
    word = "h"
    for i in range(m):
        word += "i"
    return word

def hiya(n):
    i = 1
    while i < n:
        print(hiii(i))
        i *= 2
```

$\Theta(n^2)$.

Solution: We can determine the efficiency by approximately counting the number of characters we have to store upon a call to `hiya(n)`. First, let us determine the efficiency of a call `hiii(m)`. Within `hiii`'s for loop:

- When `i` is `1`, we store the string "hi", which is 2 characters.
- When `i` is `2`, we store the string "hii", which is 3 characters.

  ...
- When `i` is `m`, we store `m + 1` characters.

Adding up these values, we see that calling `hiii(m)` causes us to store on the order of m^2 characters. (The exact value is $\frac{m(m+3)}{2} = \frac{m^2}{2} + \frac{3}{2}m$, but we really only care about the highest order term.)

Now, when we make a call `hiya(n)`, we will make calls to `hiii(1)`, `hiii(2)`, `hiii(4)`, ..., `hiii(4)`. This will store approximately $1^2 + 2^2 + 4^2 + 8^2 + ... + n^2$ characters. Calculating out the partial sums of this sequence shows that

$$1^2 = 1$$

$$1^2 + 2^2 = 5 < 2 \cdot 2^2$$

$$1^2 + 2^2 + 4^2 = 21 < 2 \cdot 4^2$$

$$1^2 + 2^2 + 4^2 + 8^2 = 85 < 2 \cdot 8^2$$

At some point, we are reasonably convinced that this pattern holds. Thus the value of $1^2 + 2^2 + 4^2 + 8^2 + ... + n^2$ is approximately n^2, within a constant factor. So we store about n^2 characters upon a call to `hiya(n)`, which means the efficiency is $\Theta(n^2)$.

Let's use OOP design to help us create a supermarket chain (think Costco)! There are many different ways to implement such a system, so there is no concrete answer.

5. What classes should we consider having? How should each of these classes interact with each other?

   There are many ways of approaching this, but one way is to have a Supermarket class to represent the entire store, an Item class to represent a certain item, a Food class to represent an item that is a food (inherits from Item), and maybe a Customer class to represent someone buying items from that store.

6. For each class, what instance and class variables would it have?

   1. Supermarket – we might have instance variables such as profit, store name, location, and a list of the items in that store along with their quantity. Note that we prefer to store the quantity inside the Supermarket, since an Item might belong to multiple Supermarkets, and each Supermarket will have a separate quantity. We might even have a price associated with each item, since specific supermarkets may mark up prices in different areas.
   2. Item – we might have instance variables such as the name and the base price.
   3. Food – we will have it inherit of all the instance variables of the Item, and also whether it is yummy, maybe the food group it is in or the expiration date.
   4. Customer – we might have some personal information, the supermarket that they're buying from, and the history of their
   5. There are some details that have been missed as well! For example, not just food items expire. Feel free to just discuss this.

7. For each class, what class methods would they have? How would they interact with each other?

   1. Once again, these are just suggestions:
   2. Supermarket
      - `check_quantity(Item)`: looks up the available quantity of that item
      - `checkout_items(Customer)`: returns the total sum of items in a customer's shopping cart, and clears their shopping cart
   3. Item
      - `check_quantity(Supermarket)`: calls `supermarket.check_quantity(self)`
   4. Food
      - `time_to_expire()`: returns an integer representing how many days before this item expires
      - `is_yummy()`: returns a boolean value of whether this item is yummy or not!
   5. Customer
      - `enter(Supermarket)`: create a shopping cart for customer in this supermarket, if it doesn't already exist
      - `leave(Supermarket)`: clear customer's shopping cart
      - `buy_item(Item)`: add item to customer's shopping cart
      - `checkout_items()`: calls `supermarket.checkout_items(Customer)`

8. Write a generator that, given m (the amount of money you have), pc (the cost of one pear), and ac (the cost of one apple) yields all possible combinations of fruit that you can buy that uses up the most of your money. In other words, each combination of fruits should not result in enough money left over to buy another fruit. Combinations of the same number of each fruit in different orders is okay. It is also okay if each combination has an extra space at the end.

```
def fruitOptions(m, pc, ac):
    """
    >>> print(list(fruitOptions(10, 2, 5)))
    ['pear pear pear pear pear ', 'pear pear apple ', 'pear apple pear
       ',
     'apple pear pear ', 'apple apple ']
    """
    if ____________________________________:
        yield ""
    if m >= pc:
        for ____________________________________________:
            ____________________________________
    if m >= ac:
        for ____________________________________________:
            ____________________________________________


    def fruitOptions(m, pc, ac):
        if m < pc and m < ac:
            yield ""
        if m >= pc:
            for p in fruitOptions(m-pc, pc, ac):
                yield "pear " + p;
        if m >= ac:
            for a in fruitOptions(m-ac, pc, ac):
                yield "apple " + a;
```

9. WWPD? Write what this python program will print.

```
a = [1, 'A', 'B', 'C', 5, 6, 7, 'D', 'E']
x = iter(a)
for i in range(5 - next(x)):
    next(x)
print(next(x))
y = iter(a)
print(next(y))
z = iter(y)
print(next(z))
```

```
6
1
A
```

10. Find an input to the year function that prints the following output: 2 0 2 5

```
def year(a):
    x = iter(a)
    y = iter(a)
    z = iter(x)
    for i in range(next(x)):
        y = iter(a)
        next(y)
    print(next(x))
    print(next(z))
    print(next(y))
    print(next(z))
```

```
year([__, 2, 0, 5, _____])
There can be any number in the first blank, and there can be any
   number of values after the 5.
```

11. DNA carries the genetic instructions that enable the functioning of many living creatures, including us. The bases of a DNA sequence include adenine (A), guanine (G), cytosine (C), and thymine (T). Adenine (A) pairs with thymine (T), and guanine (G) pairs with cytosine (C).

    Let us represent DNA as a linked list with values representing A, G, C, and T.

    Implement `reverse`, which takes in a linked list `strand` that represents a DNA strand. It destructively alters the linked list to reverse it. This function does not return anything.

```
def reverse(strand):
    """Reverses a DNA strand
    >>> d = Link("C", Link("A", Link("C", Link("G")))) \# <C A C G>
    >>> reverse(d)
    >>> print(d)
    <G C A C>
    """
    assert isinstance(strand, Link)
    if _________________:
        return ______
    reverse(_____________)
    ____________________
    ____________________
    return strand
```

```
def reverse(strand):
    """Reverses a DNA strand
    >>> d = Link("C", Link("A", Link("C", Link("G"))))  \# <C A C G>
    >>> reverse(d)
    >>> print(d)
    <G C A C>
    """
    assert isinstance(strand, Link)
    if strand is Link.empty or strand.rest is Link.empty:
        return strand
    reverse(strand.rest)
    strand.rest.rest = strand
    strand.rest = Link.empty
    return strand
```

12. Implement *isEqual*, which takes in two linked lists *strand1* and *strand2* that each represent a DNA strand. Return true if both strands are the same and false if they differ.

```
def isEqual(strand1, strand2):
    """Returns if the two strands are equal
    >>> d = Link("C", Link("A", Link("C", Link("G"))))
         <C A C G>
    >>> g = Link("C", Link("A", Link("C", Link("G"))))
     <C A C G>
    >>> isEqual(d, g)
    True
    >>> f = Link("C", Link("C", Link("G")))
     <C C G>
    >>> isEequal(d, f)
    False
    >>> n = Link("C", Link("T", Link("C", Link("G"))))
     <C T C G>
    >>> isEqual(d, n)
    False
    """
    assert isinstance(strand1, Link)
    assert isinstance(strand2, Link)
```

_____

_____

_____

_____

_____

_____

_____

Recursive Solution def isEqual(strand1, strand2): if strand1 is Link.empty and strand2 is Link.empty: return True if strand1 is Link.empty or strand2 is Link.empty: return False if strand1.first != strand2.first: return False return isEqual(strand1.rest, strand2.rest)

Iterative Solution def isEqual(strand1, strand2): while strand1 is not Link.empty and strand2 is not Link.empty: if strand1.first != strand2.first: return False strand1 = strand1.rest strand2 = strand2.rest return strand1 is Link.empty and strand2 is Link.empty

13. A frameshift mutation causes a DNA strand to shift by n nucleotides. For example, if the original DNA strand is ATTGCGA, the strand mutated by two nucleotides would be TGCGA.

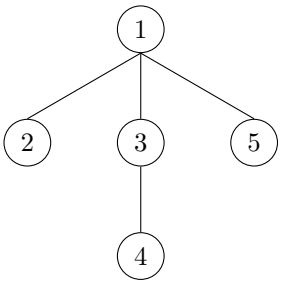
    Implement *findFrameShift*, which takes in two linked lists *original* and *mutated* that each represent a DNA strand. It returns the number of nucleotides that *original* has been shifted by. You can use the *isEqual* function. Assume the length of *original* is greater than the length of *mutated*.

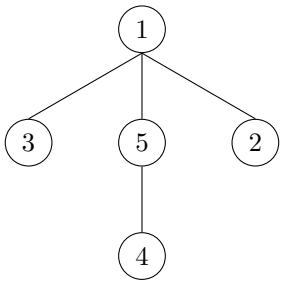
```
def findFrameShift(original, mutated):
"""Return the number of nucleotides that original has been shifted
   by after being mutated
>>> o = Link("C", Link("A", Link("C", Link("G", Link("T", Link
   ("A"))))))
 <C A C G T A>
>>> m = Link("C", Link("G", Link("T", Link ("A"))))
 <C G T A>
>>> n = findFrameshift(o,m)
>>> print(n)
2
"""
assert isinstance(original, Link)
assert isinstance(mutated, Link)
```

def findFrameShift(original, mutated): shift = 0 while mutated is not Link.empty: if isEqual(original, mutated): return shift mutated = mutated.rest shift += 1 return 0

14. Implement `rotate`, which takes in a tree and rotates the labels at each level of the tree by one to the left destructively. This rotation should be modular (That is, the leftmost label at a level will become the rightmost label after running rotate). You do NOT need to rotate across different branches.

    For example, given tree `t` on the left, `rotate(t)` should mutate `t` to give us the right.






```
def rotate(t):
  """
  >>> t1 = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(5)])
  >>> rotate(t1)
  >>> t1
  Tree(1, [Tree(3), Tree(5, [Tree(4)]), Tree(2)])
  >>> t2 = Tree(1, [Tree(2, [Tree(3), Tree(4)]),
                    Tree(5, [Tree(6)])])
  >>> rotate(t2)
  >>> t2
  Tree(1, [Tree(5, [Tree(4), Tree(3)]),
                    Tree(2, [Tree(6)])])
  """
  branch_labels = ____________________________________

  n = len(t.branches)

  for ____________________________________________:

    ____________________________________________________

    ____________________________________________________

    ____________________________________________________
```

```
def rotate(t):
  branch_labels = [b.label for b in t.branches]
  n = len(t.branches)
  for i in range(n):
    branch = t.branches[i]
    branch.label = branch_labels[(i + 1) % n]
    rotate(branch)
```

15. Implement `best_study_spot` which takes in a tree `t` full of attributes about different study spots on campus, and outputs the most ideal study spot depending on a function **`key`** that you pass in. For example, if you have a tree consisting of the distances of different spots from your dorm and you want to find the closest study spot, **`key`** would be set to min.

```
def best_study_spot(t, key):
    """Return the node in t that corresponds to the maximum value for key
       without using min or max.

    >>> t = Tree(7, [Tree(5, [Tree(9)]), Tree(3), Tree(10, [Tree(4)])])
    >>> best_study_spot(t, key=lambda x: x)
    10
    >>> best_study_spot(t, key=lambda x: -x)
    3
    >>> best_study_spot(t, key=lambda x: -abs(x - 4))
    4
    """
    if t._________():
        return _________
    best = _________
    for b in _________:
        candidate = best_study_spot(b, key)
        if _________(candidate) > key(best):
            _________ = candidate
    return _________
```

```
def best_study_spot(t, key):
    if t.is_leaf():
        return t.label
    best = t.label
    for b in t.branches:
        candidate = best_study_spot(b, key)
        if key(candidate) > key(best):
            best = candidate
    return best
```

16. You're part of a company that builds armies of robot clones from a single robot. When a robot is cloned, you can create two copies of it. The company uses trees to track how many robots are descended from each other. Implement `is_clone`, which takes in a tree `t` and checks if the tree is equally balanced on both sides.

```
def is_clone(t):
    """Return True if t is an exactly balanced tree and False if not.

    >>> t1 = Tree(1)
    >>> is_clone(t1)
    True
    >>> t2 = Tree(1, [Tree(2), Tree(3)])
    >>> is_clone(t2)
    True
    >>> t3 = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3)])
    >>> is_clone(t3)
    False
    """
    if t.is_leaf():
        return True
    if ____________:
        return False

    left, right = t.branches
    if not (is_clone(left) and is_clone(right)):
        return False

    def count_leaves(t):
        if t.is_leaf():
            return 1
        return sum(_________)

    return count_leaves(left) == ___________


def is_clone(t):
    if t.is_leaf():
        return True
    if len(t.branches) != 2:
        return False
    left, right = t.branches
    if not (is_clone(left) and is_clone(right)):
        return False
    def count_leaves(t):
        if t.is_leaf():
            return 1
        return sum(count_leaves(b) for b in t.branches)
    return count_leaves(left) == count_leaves(right)
```

17. Your teacher hides hints for the final exam in trees. However, they give you too many trees that it's hard for you to manually go through them and search for the hints. Implement a function `find_hint` that mutates a tree `t` so that it only keeps the path which does not end in a leaf node whose label is "Blank".

```
def find_hint(t):
    """Mutates the tree t so that it only keeps paths that do NOT end in a
       leaf labeled "Blank".

    >>> t1 = Tree("Start", [Tree("A", [Tree("Blank")]), Tree("B",
       [Tree("C")]), Tree("Blank")])
    >>> find_hint(t1)
    >>> print(t1)
    Tree('Start', [Tree('B', [Tree('C')])])
    >>> t2 = Tree("Start", [Tree("A", [Tree("B", [Tree("Blank")])]),
       Tree("X", [Tree("Y")])])
    >>> find_hint(t2)
    >>> print(t2)
    Tree('Start', [Tree('X', [Tree('Y')])])
    """
    for b in t.branches:
        find_hint(b)
    t.branches = [b for b in t.branches if not (b.is_leaf() and b.label ==
       ____________________)]
```

```
def find_hint(t):
    for b in t.branches:
        find_hint(b)
    t.branches = [b for b in t.branches if not (b.is_leaf() and b.label ==
       "Blank")]
```

18. Star-Lord is cruising through space and can't afford to crash into any asteroids along the way. Let his path be represented as a (possibly nested) list of integers, where an asteroid is denoted with a 0, and stars and planets otherwise. Every time Star-lord sees (visits) an asteroid (0), he merges the next planet/star with the asteroid. In other words, construct a NEW list so that all asteroids (0s) are replaced with a list containing the planet followed by the asteroid (e.g. (planet 0)). You can assume that the last object in the path is not an asteroid (0).

```
;Doctests
scm> (collision (list 1 2 3 0 4))
(1 2 3 (4 0))
scm> (collision (list 4 3 (list 0 1) 2))
(4 3 ((1 0)) 2)
scm> (collision (list 1 -2 0 -3 4 0 -5 6))
(1 -2 (-3 0) 4 (-5 0) 6)
scm> (collision (list 1 0 0 2 3))
(1 (0 0) 2 3)

;Asteroids can merge with other asteroids too

(define (collision lst)

  (cond ((_____________________) lst)

    ((_____________________________)

      _____________________________________)

    ((_____________________________)

      (cons _____________________________

        _____________________________________))

    (else _____________________________________)
  )
)
```

```
(define (collision lst)
  (cond ((null? lst) nil)
    ((list? (car lst))
      (cons (collision (car lst)) (collision (cdr lst))))
    ((and (equal? (car lst) 0) (not (null? (cdr lst))))
      (cons (list (car (cdr lst)) (car lst))
        (collision (cdr (cdr lst)))))
    (else(cons (car lst) (collision (cdr lst))))
  )
)

#Alternate solution (No cond form)

(define (collision lst)
  (if (null? lst)
    lst
    (if (list? (car lst))
      (cons (collision (car lst)) (collision (cdr lst)))
      (if (equal? (car lst) 0)
        (cons (list (cadr list) (car lst)) (collision (cddr lst)))
        (cons (car lst) (collision (cdr lst)))
      )
    )
  )
)
```

19. Write a function plan-coffee-tour that takes two lists of Berkeley coffee shops and creates an optimized tour according to the following rules: The function creates a tour by alternating shops from each list (similar to interleaving) If a coffee shop appears in both lists, it should only be visited once in the tour at its first occurrence If one list is longer than the other, the remaining unique shops should be added to the end of the tour

```
scm> (plan-coffee-tour '(binge strada philz) '(philz blue-bottle
   binge))
(binge philz strada blue-bottle)

scm> (plan-coffee-tour '(strada mind peets) '(elaichi-co philz))
(strada elaichi-co mind philz peets)

scm> (plan-coffee-tour '(strada qargo) '(strada qargo peets))
(strada qargo peets)

scm> (plan-coffee-tour '() '(delah signal))
(delah signal)
```

```
(define (plan-coffee-tour lst1 lst2)
    (cond ((_____________________________) lst2)
        ((_____________________________) lst1)
        (else
        (let ((first (car lst1))
                (rest1 (cdr lst1))
                (rest2 (_____________________________)))
            (if (_____________________________)
                (cons first (plan-coffee-tour rest1 rest2))
                (cons first (cons (_____________________________)
                                  (plan-coffee-tour rest1
                                     (_____________________________)))))))))
```

```
(define (plan-coffee-tour lst1 lst2)
    (cond ((null? lst1) lst2)
        ((null? lst2) lst1)
        (else
            (let ((first (car lst1))
                (rest1 (cdr lst1))
                (rest2 (filter (lambda (shop) (not (eq? shop
                   (car lst1)))) lst2)))
            (if (null? rest2)
                (cons first (plan-coffee-tour rest1 rest2))
                (cons first (cons (car rest2)
                                  (plan-coffee-tour rest1
                                     (cdr rest2)))))))))
```

20. Implement the macro unless, which takes a condition and a single expression. It evaluates the expression only if the condition is false.

```
scm> (unless #f (print 'nope))
nope
scm> (unless #t (print 'nope))
; nothing is printed

(define-macro (unless condition body)
    (_____________________________))
```

```
(define-macro (unless condition body)
    (list 'if (list 'not condition) body))
```

21. You're writing a plot outline for a fantasy novel. Each plot point consists of a character and an event they're involved in. Define a function build-plot-outline that takes two lists: A list of characters (symbols), A list of events (symbols)

    The function should return a list of plot points, where each plot point is a pair (character . event), following these rules:

    Plot points are created by pairing the first character with the first event, the second with the second, and so on. If one list is longer than the other, ignore the extra elements. If any character or event is 'plot-hole, skip that pairing entirely.

```
scm> (build-plot-outline '(hero villain plot-hole bard) '(battle
   scheme rescue plot-hole))
((hero . battle) (villain . scheme) (bard . rescue))

scm> (build-plot-outline '(dragon knight) '(flight plot-hole))
((dragon . flight))

scm> (build-plot-outline '(plot-hole) '(plot-hole))
()

scm> (build-plot-outline '() '(event1 event2))
()

(define (build-plot-outline characters events)
    (cond ((_____________________________) '())
            (else
            (let ((c (car characters))
                (e (car events)))
            (if (_____________________________)
                (build-plot-outline (_____________________________)
                   (_____________________________))
                (cons (_____________________________)
                        (build-plot-outline
                            (_____________________________)
                            (_____________________________)))))))))
```

```
(define (build-plot-outline characters events)
(cond ((or (null? characters) (null? events)) '())
        (else
        (let ((c (car characters))
            (e (car events)))
        (if (or (eq? c 'plot-hole) (eq? e 'plot-hole))
            (build-plot-outline (cdr characters) (cdr events))
            (cons (cons c e)
                    (build-plot-outline (cdr characters) (cdr
                       events)))))))))
```