

FINAL EXAM REVIEW

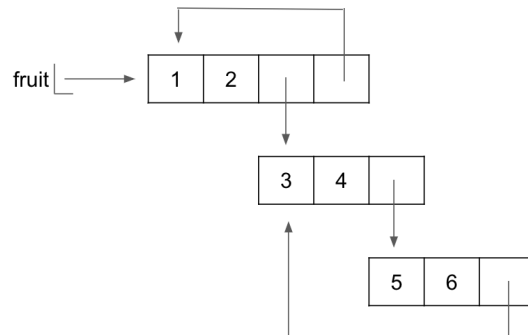
COMPUTER SCIENCE MENTORS 61A

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] = ____
```



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).

- (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)])])
    >>> list(root_to_leaf(t2))
    [[5, 5, 6]]
    """

    if _____:

        _____

    for _____:

        if _____:

            for _____:

                _____
```

- (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:

        _____
```

3 Data Abstraction

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:
            _____

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 _____
```

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=_____)
```

4 Efficiency

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
```

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?

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

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

6 Generators

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 _____:
            _____
```

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))
```

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))
```

7 LinkedLists

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
```

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>
    >>> isEqual(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)
```

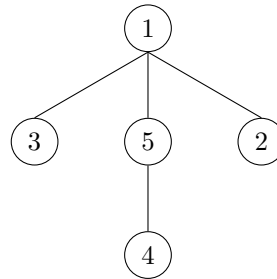
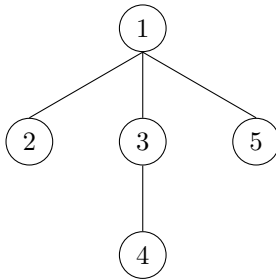
13. A frameshift mutation causes a DNA strand to shift by n nucleotides. For example, if the original DNA strand is ATTGCGA, the strand mutated by two nucleotides would be TGCGA.

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)
```

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 _____:
        _____
        _____
        _____
```

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.is_leaf():
        return t
    best = None
    for b in t.children:
        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) == _____
```

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 ==
        _____)]
```

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 _____)
)
)
```

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
              (_____ )))))))))))
```

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)
  (_____))
```


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
                      (_____ )
                      (_____ ))))))))
```