Mock Exam Problems (Homework07, Nov 23)

1 Special Methods (2021A, 10pts)

In mathematics, a set is a collection of elements. Alice implements Set with following program in Python.

```
class Set:
    def __init__(self, elements):
        self.elements = elements

    def __str__(self):
        if not self.elements:
            return '{}'
        else:
            s = '{' + str(self.elements[0])
            for element in self.elements[1:]:
                s += ', ' + str(element)
            return s + '}'

    def __repr__(self):
        return 'Set(' + repr(self.elements) + ')'
```

1.1 What Would Python Display? (4pts)

What would display when evaluating the following Python code in interactive console? Fill your answer in the blank.

```
>>> s = Set([1, 2, 3])
>>> s
__________________________________________________
>>> print(s)
__________________________________________________
>>> str(s)
__________________________________________________
>>> print(repr(s))
__________________________________________________
```

1.2 Set Operations (6pts)

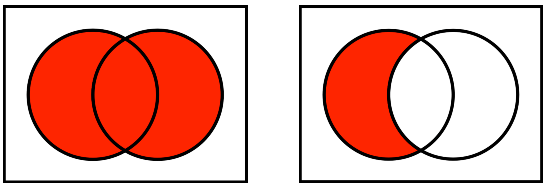
Alice discovers that her code is incorrect if some elements are duplicate. For example, `Set([1, 2, 2, 3])` produces `{1, 2, 2, 3}`, which is not a valid set because it should only contain **unique** elements.

To make elements in a set unique, Bob helps Alice rewrite the `__init__` special method in below, so that a `Set` never contains duplicated elements. Alice feels happy about that.

Besides that, Bob also writes two special methods to support addition and substraction on sets.

- The addition of two sets A, B is the set of all elements that are either members of A or members of B.
- The subtraction of two sets A, B is the set of all elements that are members of A, but not members of B.

Bob provides the following Venn diagrams to help Alice understand the two set operations.



Venn diagrams for set operations $A + B$ and $A - B$

Fill in the blanks to complete Bob's program.

```
class Set:
    def __init__(self, elements):
        self.elements = []
        for element in elements:
            if element not in self.elements:
                self.elements.append(element)

    def __add__(self, other):
        return Set(________________________ + ________________________)

    def __sub__(self, other):
        elements = []
        for _______________________________________________:
            if _______________________________________________:
                ______________________________________________
        return Set(________________)
```

2 Linked Lists (2021A, 6pts)

Fill in the blanks to implement two **in-place** mutations on linked lists:

- *reverse* takes a non-empty linked list, reverses the order of it, and returns the reversed list.
- *merge* takes two non-empty linked lists of the same length (assuming $\langle x_1, x_2, ..., x_n \rangle$ and $\langle y_1, y_2, ..., y_n \rangle$), and merges the latter one into the former one, which results to $\langle x_1, y_1, x_2, y_2, ..., x_n, y_n \rangle$.

The two functions should modify the given lists **in-place**, which means you cannot call `Link` constructor in your solution. Doctests are provided to clarify the usage and expected behavior.

```
def reverse(lnk):
    """Reverse a linked list.
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5, Link(6))))))
    >>> print(reverse(lnk))
    <6 5 4 3 2 1>
    """
    if ________________ is Link.empty:
        return lnk
    result = reverse(lnk.rest)
    ________________________________________________
    ________________________________________________
    return result

def merge(lnk1, lnk2):
    """Merge two linked lists with same length.
    >>> lnk1, lnk2 = Link(1, Link(2, Link(3))), Link(4, Link(5, Link(6)))
    >>> merge(lnk1, lnk2)
    >>> print(lnk1)
    <1 4 2 5 3 6>
    """
    while lnk1 is not Link.empty:
        rst1, rst2 = lnk1.rest, lnk2.rest
        lnk1.rest, ________________ =  ________________, ________________
        lnk1, lnk2 = rst1, rst2
```

2. (9 points) Special Methods

In mathematics, a complex number is a number that can be expressed in the form $a + bi$, where a and b are real numbers, and i represents the imaginary unit that satisfies the equation $i^2 = -1$. For example, $2 + 3i$ is a complex number. For the complex number $a + bi$, a is called the **real part**, and b is called the **imaginary part**. To emphasize, the imaginary part does not include a factor i; that is, the imaginary part is b, not bi.

Complex numbers can be added and multiplied. For any complex number c_1 and c_2 where $c_1 = a + bi$ and $c_2 = c + di$, the addition operator '+' is defined as $c_1 + c_2 = (a + c) + (b + d)i$ and the multiplication operator '·' is defined as $c_1 \cdot c_2 = (ac - bd) + (ad + bc)i$.

In the following questions, we will use Python to represent and compute complex numbers!

(a, 4 points) The code below defines a `Complex` class that represents the complex number. The instance attributes `real` and `imag` represents the real part and imaginary part of a complex number correspondingly. Read the definition carefully and for each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated.

```
class Complex:

    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __repr__(self):
        return 'So complex'

    def __str__(self):
        return 'Happy new year'
```

Expression	Interactive Output
`>>> c = Complex(1, 2)`	
`>>> c`	________________
`>>> print(c)`	________________
`>>> repr(c)`	________________
`>>> str(c)`	________________

(b, 2 points) The result of `print(c)` above looks interesting. However, as a programmer, we need a more informative `__str__` method to help us know the value of a complex number instance. Please redefine the `__str__` method of the Complex class so that the expression `print(Complex(a, b))` will present us `a+bi` on the terminal. Note that the 'a' and 'b' in `a+bi` should be replaced by the string format of the variable 'a' and 'b' from `print(Complex(a, b))`. The string format of a variable `x` can be obtained by calling `str(x)`. The doctests below may do you some favor as to understand the problem.

```
class Complex:
    ...
    def __str__(self):
        """
        >>> print(Complex(3, 4))
        3+4i
        >>> print(Complex(2.0, 0))
        2.0+0i
        >>> print(Complex(0, 1))
        0+1i
        """
        return ______________________________________________________
```

(c, 3 points) Now let's implement the addition and multiplication of complex numbers. Do you still remember the operator overloading in Python? Recall that when Python evaluates the expression `a + b`, it is in fact evaluating `a.__add__(b)`. As a result, we can define the `__add__` method in a class to change the behavior of the '+' operator, which is so called operator overloading. This feature is useful for our `Complex` class because we can write the more intuitive code `a + b` instead of `add(a, b)` when we want to express the addition of two instances of the `Complex` class, i.e. `a` and `b`. We have shown you the implementation of the `__add__` method of the `Complex` class below as an example, which satisfies the definition of the complex number's addition operator '+' mentioned above. Your task is to implement the `__mul__` method, which overloads the '*' operator in python, to satisfy the definition of the multiplication operator'·'(After implementing it, you can now easily calculate the multiplication of complex numbers and show the result by a simple Python expression `print(a * b)`).

```
class Complex:
    ...
    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.imag)

    def __mul__(self, other):
        return ________(_____________________________________________________,

                        _____________________________________________________)
```

3. (12 points) Linked List & Tree

In this problem, we have a `Link` class and a `Tree` class to use, which are defined as below.

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'




class Tree:

    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)
```

(a, 5 points) Write a function `interleave` that takes in two linked lists `link1` and `link2` and returns a **new** linked list, which is the result of `link1` and `link2` interleaved in pairs (first the node of `link1`, then the node of `link2`). If `link1` has more nodes than `link2`, just copy the remaining nodes to the new linked list, and the same is true for the condition that `link2` has more nodes than `link1`. Fill in the lines and implement the function in a recursive manner.

```
def interleave(link1, link2):
    """
    >>> link1 = Link(1, Link(3))
    >>> link2 = Link(2, Link(4, Link(6)))
    >>> link3 = interleave(link1, link2)
    >>> link3
    Link(1, Link(2, Link(3, Link(4, Link(6)))))
    >>> link3 is not link1 # should create a new linked list
    True
    """
    if link1 is Link.empty and link2 is Link.empty:
        return ______________________________________________
    if link1 is Link.empty:
        return ______________________________________________
    if link2 is Link.empty:
        return ______________________________________________
    return ____________________________________________________________________
```

(b, 7 points) Define a function `nondecreasing_paths` which takes in a nonempty Tree `t` and returns all the **nondecreasing** paths of `t`. A path is a list of labels of nodes passed **from the root** to an end node (the end node can be any node, including the root, leaves, and intermediate nodes), and a path can be denoted as $[l_1, l_2, \ldots, l_n]$, where l_i is the label of the i-th node of this path (l_1 is the root node). A nondecreasing path is a path that satisfies the requirement that for any two nodes i and j (where $i < j$) in the path, we have $l_i \le l_j$. Taking the example in the following doctests, `[2, 2, 3]` is a nondecreasing path of `t1`, while `[2, 2, 1]` is not a nondecreasing path of `t1` because the label of the third node (i.e., `1`) is smaller than that of the second node (i.e., `2`). Different paths can be in any order. We have provided a (partial) skeleton for you, and you can **either use it or not**, which does not affect your score of this question.

```
def nondecreasing_paths(t):
    """
    >>> t1 = Tree(2, [Tree(2, [Tree(3), Tree(1, [Tree(6)]), Tree(5)]), Tree(5)])
    >>> sorted(nondecreasing_paths(t1))
    [[2], [2, 2], [2, 2, 3], [2, 2, 5], [2, 5]]
    """
    # Choice 1: you can use this skeleton and fill in the lines.
    assert t
    paths = ______________________
    for ______________________ in ______________________:
        if __________________________________________:
            for ______________________ in ______________________:
                paths.append(____________________________________________)
    return paths

    # Choice 2: you can also implement this function from scratch. No penalty for
    # this problem
```

4. (15 points) Scheme

(a, 4 points) Something you should know about Scheme

(1) The two types of expressions in scheme are ________________ and ________________;

(2) Give the names of at least two special-form expressions: ____________________________;

(3) The full name of REPL is ____.