1. (12 points) World Cup

(a) **(10 pt)** For each of the expressions in the tables below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines.

Whenever the interpreter would report an error, write ERROR. You *should* include any lines displayed before an error.

Reminder: the interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

The first three rows have been provided as examples.

Assume that you have started Python 3 and executed the following statements:

```
def square(x):
    return x * x

def argentina(n):
    print(n)
    if n > 0:
        return lambda k: k(n+1)
    else:
        return 1 / n

def germany(n):
    if n > 1:
        print('hallo')
    if argentina(n-2) >= 0:
        print('bye')
    return argentina(n+2)
```

Expression	Interactive Output
`5*5`	25
`print(5)`	5
`1/0`	ERROR
`print(1, print(2))`	
`argentina(0)`	

Expression	Interactive Output
`argentina(1)(square)`	
`germany(1)(square)`	
`germany(2)(germany)`	

(b) **(2 pt)** Fill in the blank with an expression so that the whole expression below evaluates to a number.
Hint: The expression `abs > 0` causes a `TypeError`.

```
(lambda t: argentina(t)(germany)(square))(________________________________)
```

3. (8 points) Express Yourself

(a) (3 pt) A k-bonacci sequence starts with K-1 zeros and then a one. Each subsequent element is the sum of the previous K elements. The 2-bonacci sequence is the standard Fibonacci sequence. The 3-bonacci and 4-bonacci sequences each start with the following ten elements:

```
               n: 0, 1, 2, 3, 4, 5, 6,  7,  8,  9, ...

kbonacci(n, 2): 0, 1, 1, 2, 3, 5, 8, 13, 21, 35, ...

kbonacci(n, 3): 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, ...

kbonacci(n, 4): 0, 0, 0, 1, 1, 2, 4,  8, 15, 29, ...
```

Fill in the blanks of the implementation of `kbonacci` below, a function that takes non-negative integer `n` and positive integer `k` and returns element n of a k-bonacci sequence.

```
def kbonacci(n, k):
    """Return element N of a K-bonacci sequence.

    >>> kbonacci(3, 4)
    1
    >>> kbonacci(9, 4)
    29
    >>> kbonacci(4, 2)
    3
    >>> kbonacci(8, 2)
    21
    """

    if n < k - 1:

        return 0

    elif n == k - 1:

        return 1

    else:

        total = 0


        i = ____________________________________________________________________

        while i < n:


            total = total + ____________________________________________________

            i = i + 1

        return total
```

(b) **(5 pt)** Fill in the blanks of the following functions defined together in the same file. **Assume that all arguments to all of these functions are positive integers that do not contain any zero digits**. For example, 1001 contains zero digits (not allowed), but 1221 does not (allowed). You may assume that `reverse` is correct when implementing `remove`.

```
def combine(left, right):
    """Return all of LEFT's digits followed by all of RIGHT's digits."""
    factor = 1
    while factor <= right:
        factor = factor * 10
    return left * factor + right

def reverse(n):
    """Return the digits of N in reverse.

    >>> reverse(122543)
    345221
    """

    if n < 10:

        return n

    else:

        return combine(_________________________ , _________________________)


def remove(n, digit):
    """Return all digits of N that are not DIGIT, for DIGIT less than 10.

    >>> remove(243132, 3)
    2412
    >>> remove(243132, 2)
    4313
    >>> remove(remove(243132, 1), 2)
    433
    """

    removed = 0

    while n != 0:


        ____________, ____________ = ____________________, ___________________

        if ____________________________________________________________________:

            removed = _______________________________________________________

    return reverse(removed)
```

4. (6 points) Lambda at Last

(a) **(2 pt)** Fill in the blank below with an expression so that the second line evaluates to 2014. **You may only use the names `two_thousand`, `two`, `k`, `four`, and `teen` and parentheses in your expression (no numbers, operators, etc.)**.

```
two_thousand = lambda two: lambda k: __________________________________________
two_thousand(7)(lambda four: lambda teen: 2000 + four + teen)
```

(b) **(4 pt)** The `if_fn` returns a two-argument function that can be used to select among alternatives, similar to an `if` statement. Fill in the return expression of `factorial` so that it is defined correctly for non-negative arguments. **You may only use the names `if_fn`, `condition`, `a`, `b`, `n`, `factorial`, `base`, and `recursive` and parentheses in your expression (no numbers, operators, etc.)**.

```
def if_fn(condition):
    if condition:
        return lambda a, b: a
    else:
        return lambda a, b: b

def factorial(n):
    """Compute N! for non-negative N.  N! = 1 * 2 * 3 * ... * N.

    >>> factorial(3)
    6
    >>> factorial(5)
    120
    >>> factorial(0)
    1
    """
    def base():
        return 1
    def recursive():
        return n * factorial(n-1)


    return ____________________________________________________________________
```

1. (12 points) Evaluators Gonna Evaluate

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "Error".

Hint: No answer requires more than 5 lines. (It's possible that all of them require even fewer.)

The first two rows have been provided as examples.

Recall: The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

Assume that you have started `python3` and executed the following statements:

```
def jazz(hands):
    if hands < out:
        return hands * 5
    else:
        return jazz(hands // 2) + 1

def twist(shout, it, out=7):
    while shout:
        shout, out = it(shout), print(shout, out)
    return lambda out: print(shout, out)

hands, out = 2, 3
```

Expression	**Interactive Output**
`pow(2, 3)`	8
`print(4, 5) + 1`	4 5 Error
`print(None, print(None))`	
`jazz(5)`	
`(lambda out: jazz(8))(9)`	
`twist(2, lambda x: x-2)(4)`	
`twist(5, print)(out)`	
`twist(6, lambda hands: hands-out, 2)(-1)`	

3. (10 points) Digit Fidget

****IMPORTANT DEFINITION**** Each digit in a non-negative integer `n` has a *digit position.* Digit positions begin at 0 and count from the right-most digit of `n`. For example, in 568789, the digit 9 is at position 0 and digit 7 is at position 2. The digit 8 appears at both positions 1 and 3.

(a) **(3 pt)** Implement the `find_digit` function, which takes a non-negative integer `n` and a digit `d` **greater than 0 and less than 10**. It returns the *largest* (left-most) position in `n` at which digit `d` appears. If `d` does not appear in `n`, then `find_digit` returns `False`. *You may not use recursive calls.*

```
def find_digit(n, d):
    """Return the largest digit position in n for which d is the digit.

    >>> find_digit(567, 7)
    0
    >>> find_digit(567, 5)
    2
    >>> find_digit(567, 9)
    False
    >>> find_digit(568789, 8)  # 8 appears at positions 1 and 3
    3
    """

    i, k = 0, ____________________________________________________________

    while n:

        n, last = n // 10, n % 10

        if last == ______________________________________________________:

            _____________________________________________________________

        i = i + 1

    return ______________________________________________________________
```

(b) **(2 pt)** Circle **all** values of y for which the final expression below evaluates to `True`. Assume that `find_digit` is implemented correctly. The `compose1` function appears on the left column of page 2 of your study guide.

1   2   3   4   5   6   7   8   9

```
f = lambda x: find_digit(234567, x)
compose1(f, f)(y) == y
```

(c) (3 pt) Implement `luhn_sum`. The *Luhn sum* of a non-negative integer n adds the sum of each digit in an *even* position to the sum of doubling each digit in an *odd* position. If doubling an odd digit results in a two-digit number, those two digits are summed to form a single digit. *You may not use recursive calls or call `find_digit` in your solution.*

```
def luhn_sum(n):
    """Return the Luhn sum of n.

    >>> luhn_sum(135)     # 1 + 6 + 5
    12
    >>> luhn_sum(185)     # 1 + (1+6) + 5
    13
    >>> luhn_sum(138743)  # From lecture: 2 + 3 + (1+6) + 7 + 8 + 3
    30
    """
    def luhn_digit(digit):

        x = digit * ________________________________________________________

        return (x // 10) + _________________________________________________

    total, multiplier = 0, 1

    while n:

        n, last = n // 10, n % 10

        total = total + luhn_digit(last)

        multiplier = __________________________________________ - multiplier

    return total
```

(d) (2 pt) A non-negative integer has a *valid* Luhn sum if its Luhn sum is a multiple of 10. Implement `check_digit`, which appends one additional digit to the end of its argument so that the result has a valid Luhn sum. Assume that `luhn_sum` is implemented correctly.

```
def check_digit(n):
    """Add a digit to the end of n so that the result has a valid Luhn sum.

    >>> check_digit(153) # 2 + 5 + 6 + 7 = 20
    1537
    >>> check_digit(13874)
    138743
    """

    return _________________________________________________________________
```

3. (14 points) You Complete Me

(a) **(4 pt)** Implement the `longest_increasing_suffix` function, which returns the longest suffix (end) of a positive integer that consists of strictly increasing digits.

```
def longest_increasing_suffix(n):
    """Return the longest increasing suffix of a positive integer n.

    >>> longest_increasing_suffix(63134)
    134
    >>> longest_increasing_suffix(233)
    3
    >>> longest_increasing_suffix(5689)
    5689
    >>> longest_increasing_suffix(568901) # 01 is the suffix, displayed as 1
    1
    """

    m, suffix, k = 10, 0, 1

    while n:

        ______________________________________________, last = n // 10, n % 10

        if ___________________________________________________________________:

            m, suffix, k = ______________, ___________________________, 10 * k

        else:

            return suffix

    return suffix
```

(b) **(3 pt)** Add parentheses and single-digit integers in the blanks below so that the expression on the second line evaluates to 2015. **You may only add parentheses and single-digit integers.** You may leave some blanks empty.

```
lamb = lambda lamb: lambda: lamb + lamb


lamb(1000)______ + (lambda b, c: b______  *  b______  -  c______)(lamb(______), 1)______
```

(c) **(3 pt)** Implement the `combine` function, which takes a non-negative integer `n`, a two-argument function `f`, and a number `result`. It applies `f` to the first digit of `n` and the result of combining the rest of the digits of `n` by repeatedly applying `f` (see the doctests). If `n` has no digits (because it is zero), `combine` returns `result`.

```
from operator import add, mul

def combine(n, f, result):
    """Combine the digits in non-negative integer n using f.

    >>> combine(3, mul, 2) # mul(3, 2)
    6
    >>> combine(43, mul, 2) # mul(4, mul(3, 2))
    24
    >>> combine(6502, add, 3) # add(6, add(5, add(0, add(2, 3))))
    16
    >>> combine(239, pow, 0) # pow(2, pow(3, pow(9, 0)))
    8
    """
    if n == 0:

        return result

    else:

        return combine(___________, ___________, _______________________________)
```

(d) **(4 pt)** Implement the `memory` function, which takes a number `x` and a single-argument function `f`. It returns a function with a peculiar behavior that you must discover from the doctests. **You may only use names and call expressions in your solution. You may not write numbers or use features of Python not yet covered in the course.**

```
square = lambda x: x * x
double = lambda x: 2 * x

def memory(x, f):
    """Return a higher-order function that prints its memories.

    >>> f = memory(3, lambda x: x)
    >>> f = f(square)
    3
    >>> f = f(double)
    9
    >>> f = f(print)
    6
    >>> f = f(square)
    3
    None
    """
    def g(h):

        print(_____________________________________________________________)


        return ____________________________________________________________

    return g
```

5. (4 points) Your Father's Parentheses

Suppose we have a sequence of quantities that we want to multiply together, but can only multiply two at a time. We can express the various ways of doing so by counting the number of different ways to parenthesize the sequence. For example, here are the possibilities for products of 1, 2, 3, 4 and 5 elements:

Product	a	ab	abc	abcd	abcde		
Count	1	1	2	5	14		
Parenthesizations	a	ab	a(bc)	a(b(cd))	a(b(c(de)))	(ab)(c(de))	(a((bc)d))e
			(ab)c	a((bc)d)	a(b((cd)e))	(ab)((cd)e)	((ab)(cd))e
				(ab)(cd)	a((bc)(de))	(a(bc))(de)	((a(bc))d)e
				(a(bc))d	a((b(cd))e)	((ab)c)(de)	(((ab)c)d)e
				((ab)c)d	a(((bc)d)e)	(a(b(cd)))e	

Assume, as in the table above, that we don't want to reorder elements.

Define a function `count_groupings` that takes a positive integer n and returns the number of ways of parenthesizing the product of n numbers. (You might not need to use all lines.)

```
def count_groupings(n):
    """For N >= 1, the number of distinct parenthesizations
    of a product of N items.
    >>> count_groupings(1)
    1
    >>> count_groupings(2)
    1
    >>> count_groupings(3)
    2
    >>> count_groupings(4)
    5
    >>> count_groupings(5)
    14
    """
    if n == 1:

        return ________________________________________________________________


    ____________________________________________________________________________


    i = ________________________________________________________________________


    while ____________________________________________________________:


        ________________________________________________________________________

        i += 1

    return ______________________________________________________________________
```