INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5" × 11" crib sheet of your own creation and the official CS 61A study guides.
- Mark your answers on the exam itself. We will not grade answers written on scratch paper.

<table>
<thead>
<tr>
<th>Last name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td></td>
</tr>
<tr>
<td>Student ID number</td>
<td></td>
</tr>
<tr>
<td>CalCentral email</td>
<td>(<a href="mailto:_@berkeley.edu">_@berkeley.edu</a>)</td>
</tr>
<tr>
<td>TA</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your left</td>
<td></td>
</tr>
<tr>
<td>Name of the person to your right</td>
<td></td>
</tr>
</tbody>
</table>

All the work on this exam is my own. (please sign)
1. (8 points) Halloween (At least one of these is out of Scope: WWPD, Objects, Linked Lists)

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”, but include all output displayed before the error. The Link class appears on page 2 of the midterm 2 study guide.

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:

```python
class Party:
    guests = Link.empty

    def __init__(self, time):
        Party.guests = Link(time+1, Party.guests)

    def attend(self):
        self.guests.rest = Link(self.guests.rest)
        return self.guests

class Costume(Party):
    def __init__(self, bow, tie):
        Party.guests.rest = Link(bow)
        self.ie = Link(self)

    def attend(self):
        print(repr(self.ie))
        Party.attend = lambda self: Party(9).guests

    def __repr__(self):
        print('Nice')
        return 'Costume'
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link(1, Link.empty)</td>
<td>Link(1)</td>
</tr>
<tr>
<td>Link(1, Link(2))</td>
<td>Link(1, Link(2))</td>
</tr>
<tr>
<td>Party(1).guests</td>
<td></td>
</tr>
<tr>
<td>Party(3).attend()</td>
<td></td>
</tr>
<tr>
<td>Costume(5, 6).attend()</td>
<td></td>
</tr>
<tr>
<td>Party(7).attend()</td>
<td></td>
</tr>
</tbody>
</table>
2. (8 points) **A List with a Twist** *(At least one of these is out of Scope: Environment Diagram, Nonlocal, Lists)*

Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.
- Use box-and-pointer notation for list values. You do not need to write index numbers or the word “list”.

```python
lamb = 'da'
def da(da):
    def lamb(lamb):
        nonlocal da
        def da(nk):
            da = nk + ['da']
            da.append(nk[0:2])
            return nk.pop()
da(lamb)
return da([[1], 2]) + 3
da(lambda da: da(lamb))
```

<table>
<thead>
<tr>
<th>Global frame</th>
<th>f1: ________ [parent=____________]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Return Value]</td>
</tr>
<tr>
<td>f2: ________ [parent=____________]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Return Value]</td>
</tr>
<tr>
<td>f3: ________ [parent=____________]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Return Value]</td>
</tr>
<tr>
<td>f4: ________ [parent=____________]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Return Value]</td>
</tr>
</tbody>
</table>
3. (6 points) Return Policy (*All are in Scope: Nonlocal, Lists, Mutability*)

Implement `quota`, which takes a one-argument function `f` and a non-negative integer `limit`. The function it returns has the same behavior as `f`, except that each value is only returned up to `limit` times. After that, the function returns an “Over quota” message instead, and the `limit` is decreased by 1 for future calls.

```python
def quota(f, limit):
    """A decorator that limits the number of times a value can be returned."

    >>> square = lambda x: x * x
    >>> square = quota(square, 3)
    >>> square(6)  # 1st call with return value 36
    36
    >>> [square(5) for x in range(3)]  # 3 calls when the limit is 3
    [25, 25, 25]
    >>> square(5)  # 4th call with return value 25
    'Over quota! Limit is now 2'
    >>> square(-6)  # 2nd call with return value 36
    36
    >>> square(-6)  # 3rd call when the limit is 2
    'Over quota! Limit is now 1'
    >>> square(7)  # 1st call when the limit is 1
    49
    >>> square(5)  # 5th call with return value 25
    'Over quota! Limit is now 0'
    ""
values = []

def limited(n):
    ________________________________
    y = ____________________________
    count = len(______________________)
    values.append(y)
    if ______________________________:
        return __________________________
    limit = __________________________
    return 'Over quota! Limit is now ' __________________
return limited
```
4. (6 points) A Classy Election (All are in Scope: Objects)

Implement the VotingMachine and Ballot classes based on the doctest below. The voting machine must determine which choice has the most votes (the \textit{winner}) and detect if a ballot is used more than once. In case of a tie, the \textit{winner} among choices with maximal votes is the one that most recently received a vote. Ballot.vote takes a string, and a VotingMachine must handle an arbitrary number of choices.

```python
class VotingMachine:
    """A machine that creates and records ballots.
    >>> machine = VotingMachine(4)
    >>> a, b, c, d = machine.ballots
    >>> d.vote('Bruin')
    'Bruin is winning'
    >>> b.vote('Bruin')
    'Bruin is winning'
    >>> c.vote('Bear')
    'Bear is losing'
    >>> a.vote('Bear')
    'Bear is winning'
    >>> c.vote('Tree')
    'Fraud: multiple votes from the same ballot!'
    >>> machine.winner
    'Bear'
    ""
    def __init__(self, k):
        self.ballots = [________________________ for i in range(k)]
        self.votes = {}

    def record(self, ballot, choice):
        if ballot.used:
            return 'Fraud: multiple votes from the same ballot!'
        ________________________________
        self.votes[choice] = ________________________________ + 1

        if ________________________________:
            return choice + ' is losing'
        else:
            ________________________________
            return choice + ' is winning'

class Ballot:
    ________________________________
    def __init__(self, machine):
        self.machine = machine
        def vote(self, x):
            return ________________________________
```
5. (6 points) **Trick or Tree** *(All are in Scope: Trees, Linked Lists)*

Implement path, which takes a linked list \( s \) and a Tree instance \( t \). It returns whether \( s \) is a path from the root of \( t \) to some leaf. The Tree and Link classes are on page 2 of the midterm 2 study guide.

**Restrictions:**

- You may not call the built-in `len` function on a linked list or invoke its `__len__` method.
- You may not apply element selection (e.g., \( s[2] \)) on a linked list or invoke its `__getitem__` method.

```python
def path(s, t):
    """Return whether Link S is a path from the root to a leaf in Tree T."
    
    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)]), Tree(6)])
    >>> a = Link(1, Link(3, Link(4)))  # A full path
    >>> path(a, t)
    True
    >>> b = Link(1, Link(3))  # A partial path
    >>> path(b, t)
    False
    >>> c = Link(1, Link(2, Link(7)))  # A path and an extra value
    >>> path(c, t)
    False
    >>> d = Link(3, Link(4))  # A path of a branch
    >>> path(d, t)
    False
    """
    if ________________________________________________________________________:
        return False
    if ________________________________________________________________________:
        return True
    return __________([_______________________________ for b in t.branches])
```

6. (6 points) Left it Right There

(a) (4 pt) (All are in Scope: Trees) Implement binary, which takes a list s of unique integers. It returns a binary search tree containing all of those integers, represented as a BTree instance or BTree.empty. The values in any path of this tree must appear in the same order as they did in s. The BTree class is on page 2 of the midterm 2 study guide.

```python
def binary(s):
    """Construct a binary search tree from S for which all paths are in order."
    assert len(s) == len(set(s)), 'All elements of s should be unique'

    if ________________________________________________________________________:
        return ________________________________________________________________

    root = ____________________________________________________________________

    left = ____________________________________________________________________

    right = ___________________________________________________________________

    return BTree(root, binary(left), binary(right))
```

(b) (1 pt) (At least one of these is out of Scope: Growth) Circle the Θ expression that describes the smallest possible height of the tree returned by binary(s) for a list s of length n. The height of a tree is the length of the longest path from its root to a leaf.

- Θ(1)
- Θ(log n)
- Θ(n)
- Θ(n²)
- Θ(2ⁿ)
- None of these

(c) (1 pt) (At least one of these is out of Scope: Growth) Circle the Θ expression that describes the largest possible height of the tree returned by binary(s) for a list s of length n. The height of a tree is the length of the longest path from its root to a leaf.

- Θ(1)
- Θ(log n)
- Θ(n)
- Θ(n²)
- Θ(2ⁿ)
- None of these
7. (10 points) Summer Camp

(a) (6 pt) (All are in Scope: Tree Recursion, Lists) Implement \texttt{sums}, which takes two positive integers \texttt{n} and \texttt{k}. It returns a list of lists containing all the ways that a list of \texttt{k} positive integers can sum to \texttt{n}. Results can appear in any order.

```python
def sums(n, k):
    """Return the ways in which \texttt{K} positive integers can sum to \texttt{N}.
    >>> sums(2, 2)
    [[1, 1]]
    >>> sums(2, 3)
    []
    >>> sums(4, 2)
    [[3, 1], [2, 2], [1, 3]]
    >>> sums(5, 3)
    [[3, 1, 1], [2, 2, 1], [1, 3, 1], [2, 1, 2], [1, 2, 2], [1, 1, 3]]
    """
    if ________________________________________________________________________:
        return _________________________________________________________________
    y = []
    for x in __________________________________________________________________:
        y.extend([______________________ for s in sums(______________________)])
    return y
```

(b) (4 pt) (All are in Scope: Lists, Lambdas) Why so many lines? Implement \texttt{f} and \texttt{g} for this alternative version of the \texttt{sums} function.

```python
f = lambda x, y: (x and [______________ for z in y] + f(_______, _______)) or []

def sums(n, k):
    """Return the ways in which \texttt{K} positive integers can sum to \texttt{N}."""
    g = lambda w: (w and f(___________________________________________)) or [[]]
    return [v for v in g(k) if sum(v) == n]
```