1. Sample Final Questions

1.1 It’s holiday season!

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.

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

```python
class Coffee:
    warmth = 2
    def __init__(self, warmth):
        self.warmth = warmth
        def holiday(self, cheer):
        def buzz(self, sugar):
            yield sugar(self, cheer) + self.warmth
        cal = self.buzz
        while True:
            yield sugar(self, next(cal)) + self.warmth
        return buzz

class Latte(Coffee):
    @property
        def buzz(self):
            return iter([[self.warmth
                for _ in range(Latte.
                warmth)])

    def stir(self, drink):
        self.warmth += 1
        return drink * drink
```
frap, mocha = Coffee(2), Latte(1)
Latte.holiday, Coffee.buzz = frap.holiday(5), frap.holiday(5)
chai = mocha.holiday(lambda self, x: x + 1)
vanilla = frap.buzz(Latte.stir)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5*5</td>
<td>25</td>
</tr>
<tr>
<td>print(5)</td>
<td>5</td>
</tr>
<tr>
<td>1/0</td>
<td>ERROR</td>
</tr>
<tr>
<td>Latte.stir(Latte, 3)</td>
<td></td>
</tr>
<tr>
<td>print(Latte.warmth, mocha.warmth)</td>
<td></td>
</tr>
<tr>
<td>next(chai)</td>
<td></td>
</tr>
<tr>
<td>next(chai)</td>
<td></td>
</tr>
<tr>
<td>[cream for cream in chai]</td>
<td></td>
</tr>
<tr>
<td>next(vanilla)</td>
<td></td>
</tr>
<tr>
<td>next(vanilla)</td>
<td></td>
</tr>
</tbody>
</table>

1.2 Lazy Newton

1. Recall the iterative improvement algorithm, where we keep updating a guess until either it is close enough to the actual answer or we have exhausted our number of attempts. What if we wanted to model an infinite sequence of updates? Modify the improve function such that it outputs a Stream of values at each update instead of just the final value. Note that max_updates can either be a number or None, which means that the Stream should be infinite.

```python
def lazy_improve(update, close, guess=1, max_updates=None):
```

1.3 Higher Order Mus

Line 1: (define f (mu (x) (mu (y) (+ x y))))
Line 2: (define (g z) (let ((z 3) (omega (f z))) (omega z)))
Line 3: (g 4)
1. Draw the complete environment diagram that results from executing the program above until the entire program is finished or an error occurs. **Hint:** do let form invokes a make call frame method. If SchemeError occurs, circle the expression whose evaluation directly causes the error.

2. How many calls to scheme_eval were made when evaluating \((g \ 3)\)?
   - 9
   - 12
   - 15
   - 18

3. I now modify Line 1 to be Line 1b: \((\text{define } f \ (\mu \ (z) \ (\mu \ (y) \ (+ \ z \ y)))\))
   What will a correct implementation of the Scheme interpreter in Project 4 output for \((g \ 4)\)? If SchemeError occurs, box the expression (from Lines 1b, 2, or 3) whose evaluation directly causes the error.

1.4 **Prepare for trouble! And make it double...**

Team Rocket has caught you in a cave and placed teleporters everywhere around you! However, they are kind enough to tell you the goal location to exit the cave, the actions you can take at a square, and the possible destination list for each result. If you are at \((2, 1)\), for instance, you know you can go “up” and the list of destinations you can possibly be at after going “up”.

Since you don’t know which destination you’ll end up in, you need to consider the fact that you can be in all of them. Formally, the problem object is a given dictionary that will map a location to a list of result objects, whose selectors are described below:

```python
# Result abstract data type
def get_direction_name(result):
   
   # Implementation is hidden"

def get_destinations(result):
```

CS 61A Fall 2014: John DeNero, with
Soumya Basu, Matthew Chow, Ajeya Cotra, Brian Hou, Andrew Huang, Robert Huang, Michelle Hwang, Mehdi Jamei, Joy Jeng, Chloe Lischinsky, Kaylee Mann, Beth Marrone, Allen Nguyen, Youri Park, Jack Qiao, Sumukh Sridhara, Steven Tang, Michael Tao, Dickson Tsai, Iris Wang, Albert Wu, Chenyang Yuan, Marvin Zhang
class InfiniteLoopException(Exception):
    pass

def solve(problem):
    return choose_action(problem['initial'], problem, [])

solve should return an action: a list of the action’s name and a plan, which is a dictionary mapping each possible destination to another appropriate action. An output to solve(team_rocket) will look something like:

['up', {(2, 2): ['left', {(1, 2): ['up', (3, 1): ['up', {(1, 1): ['goal', {}}], (3, 3): ['down', {(1, 1): ['goal', {}}]]}]}, (3, 3): ['down', {(1, 1): ['goal', {}}]]}]}

Your task is to fill in the implementations of choose_action and create_plan such that the solve function above will output an action that will always lead you to the goal destination (or if such a plan does not exist, raise an InfiniteLoopException). Additionally, your code cannot violate any abstraction barriers.

def choose_action(location, problem, path):
    if location == problem['goal']:
        return ['goal', {}]
    if location in path:
        ____________________________________________:
        try:
            ____________________________________________:
            ________________________________:
            except InfiniteLoopException:
                pass
def create_plan(locations, problem, path):
    plan = {}
    for location in locations:
        ...

    return plan

1.5 It's a (Hailstone) Race!

1. In SQL, it is possible to make a decision about what to output based on input data using the `CASE` statement. The syntax looks like:

```
SELECT
    CASE WHEN fur = "curly" THEN fur ELSE name END
AS curly_replaced FROM dogs;
```

Given the (truncated) `dogs` table from lecture, the output will then look like:

<table>
<thead>
<tr>
<th>Name</th>
<th>Fur</th>
<th>curly_replaced</th>
</tr>
</thead>
<tbody>
<tr>
<td>delano</td>
<td>long</td>
<td>delano</td>
</tr>
<tr>
<td>eisenhower</td>
<td>short</td>
<td>eisenhower</td>
</tr>
<tr>
<td>fillmore</td>
<td>curly</td>
<td>curly</td>
</tr>
<tr>
<td>grover</td>
<td>short</td>
<td>grover</td>
</tr>
<tr>
<td>Herbert</td>
<td>curly</td>
<td>curly</td>
</tr>
</tbody>
</table>

Now, given a table `ints` of two columns `a`, `b`, output a table that shows the hailstone steps for each number, stopping when any of the numbers reaches 1. With SQL, it’s easy to see the recursion side-by-side, so now, you can compare which numbers’ hailstone sequences are longer! The blanks are not indented.

-- Given: (SQL commands are case insensitive)
CREATE TABLE ints AS SELECT 20 AS a, 21 AS b;
-- First two rows of query output should then be:
-- 20 | 21
-- 10 | 64
select case when ______ then ______ else ______ end,

    case when __________ then ______ else ______________ end

1.6 Hall of Fame (HoF)

1. Given the following assignment statements, write a call expression that will evaluate
to the string “FINAL”. Your call expression itself cannot contain any strings.

bird = lambda boston: boston("N")
magic = lambda lakers: lambda: "LA" + lakers
jordan = lambda chicago: kobe(chicago+"IF")()
kobe = lambda duncan: lambda: duncan[::-1]

1.7 Trick or Trees

1. Given the following implementation of the Tree class - construct an iterator that out-
puts the value of every candy ingredient in the tree. The iterator should replace any
instances of the string “Broccoli” with “Trick!” For example:

>>> sweet = Tree("Sugar", Link(Tree("Chocolate"), Link(Tree("Broccoli"))))
>>> pb = Tree("Peanut", Link(Tree("Butter")))
>>> goodies = Tree("Broccoli", Link(pb, Link(sweet)))
>>> for treat in goodies:
...    print(treat)
Trick!
Peanut
Butter
Sugar
Chocolate
Trick!
Complete the \_\_iter\_\_ method of the Tree class. Careful with the attribute names in the Link class. Implementations may differ, but the concepts remain the same.

nil = "candy!"

**class** Link:

```python
def __init__(self, first, second=nil):
    self.value = first
    self.rest = second
```

**class** Tree:

```python
def __init__(self, entry, branches=nil):
    self.entry = entry
    self.branches = branches

def __iter__(self):
    if ________________________________________________:
        _______________________________________________
    else:
        _______________________________________________
    current = _______________________________________
    while current != _______________________________
        elem = _______________________________________
        _______________________________________________
        _______________________________________________
        _______________________________________________
        current = _____________________________________
```