CS 61A Structure and Interpretation of Computer Programs

Summer 2013

MIDTERM 2 SOLUTIONS

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 61A midterm 1 study guide attached to the back of this exam.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation.

Last name	
First name	
SID	
Login	
TA & section time	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

1. (9 points) Do you feel lucky, punk?

For each of the following expressions, write the value to which it evalutes in the interactive Python interpreter. The first two rows have been provided as examples.

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

```
lucky = [3, 4, 5, 6, 7]
beginning = [lucky, [3, 4, 5, 6, 7]]
punk = beginning[1]
```

Expression	Evaluates to
`3 + 3`	6
`1 / 0`	Error
`lucky[1]`	4
`beginning[1] == lucky`	True
`beginning[1] is lucky`	False
`punk[2:]`	[5, 6, 7]
`punk == lucky`	True

Now, assume the following line is executed *in the same interpreter session*:

```
beginning[0].append(10)
```

Expression	Evaluates to
`lucky`	[3, 4, 5, 6, 7, 10]
`beginning[0]`	[3, 4, 5, 6, 7, 10]
`beginning[1]`	[3, 4, 5, 6, 7]
`punk == lucky`	False

2. (6 points) Not with a fizzle, but with a bang

(a) (2 pt) Consider the following function definition.

```
def fizzle(n):
    if n <= 0:
        return n
    elif n % 23 == 0:
        return n
    return fizzle(n - 1)
```

Circle the correct order of growth for a call to `fizzle(n)`:

$\Theta(1)$ $\Theta(\log n)$ $\Theta(\sqrt{n})$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

(b) (2 pt) Now consider the following function definitions.

```
def boom(n):
    if n == 0:
        return "BOOM!"
    return boom(n - 1)

def explode(n):
    if n == 0:
        return boom(n)
    i = 0
    while i < n:
        boom(n)
        i += 1
    return boom(n)
```

Circle the correct order of growth for a call to `explode(n)`:

$\Theta(1)$ $\Theta(\log n)$ $\Theta(\sqrt{n})$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

(c) (2 pt) Now consider the following function definition.

```
def dreams(n):
    if n <= 0:
        return n
    if n > 0:
        return n + dreams(n // 2)
```

Circle the correct order of growth for a call to `dreams(n)`:

$\Theta(1)$ $\Theta(\log n)$ $\Theta(\sqrt{n})$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

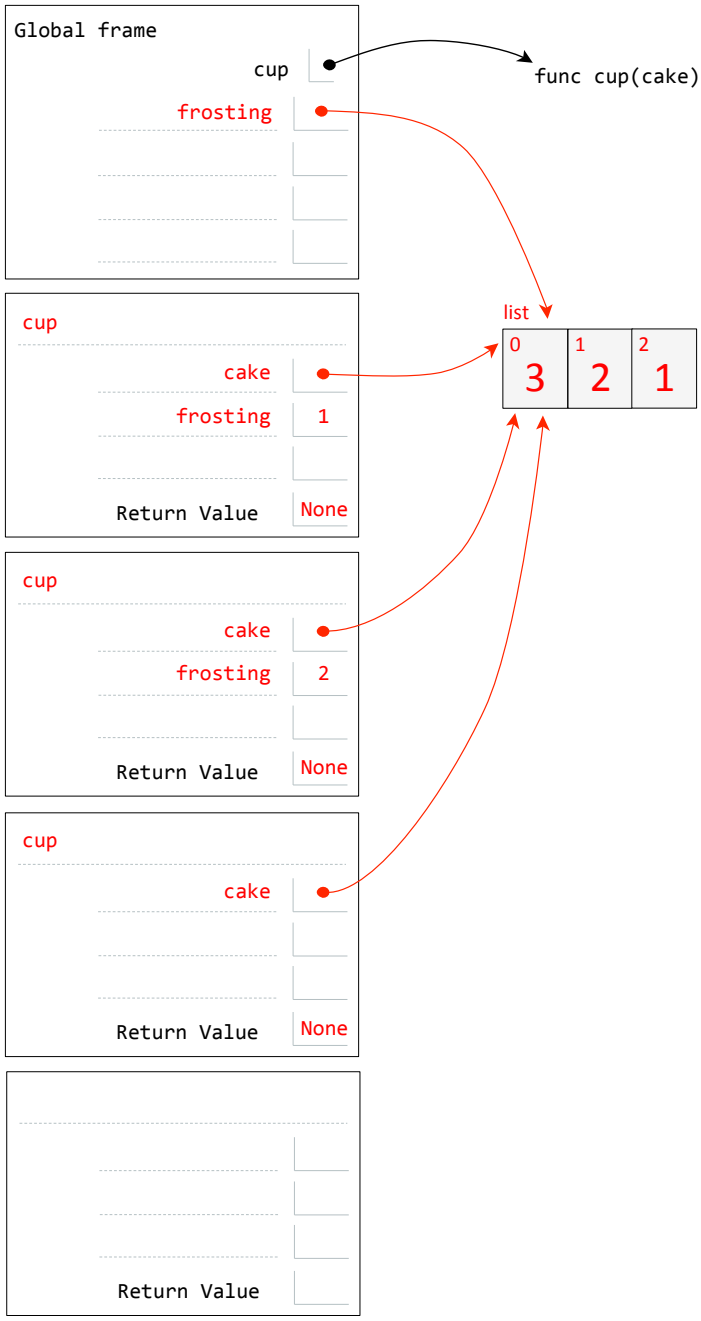
3. (12 points) Cupcakes cupcakes cupcakes cupcakes cupcakes

(a) (6 pt) 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 need only show the final state of each frame. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
def cup(cake):
    if len(cake) != 1:
        frosting = cake.pop(0)
        cup(cake)
        cake.append(frosting)
frosting = [1, 2, 3]
cup(frosting)
```



(b) **(6 pt)** 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 need only show the final state of each frame. *You may not need to use all of the spaces or frames.*

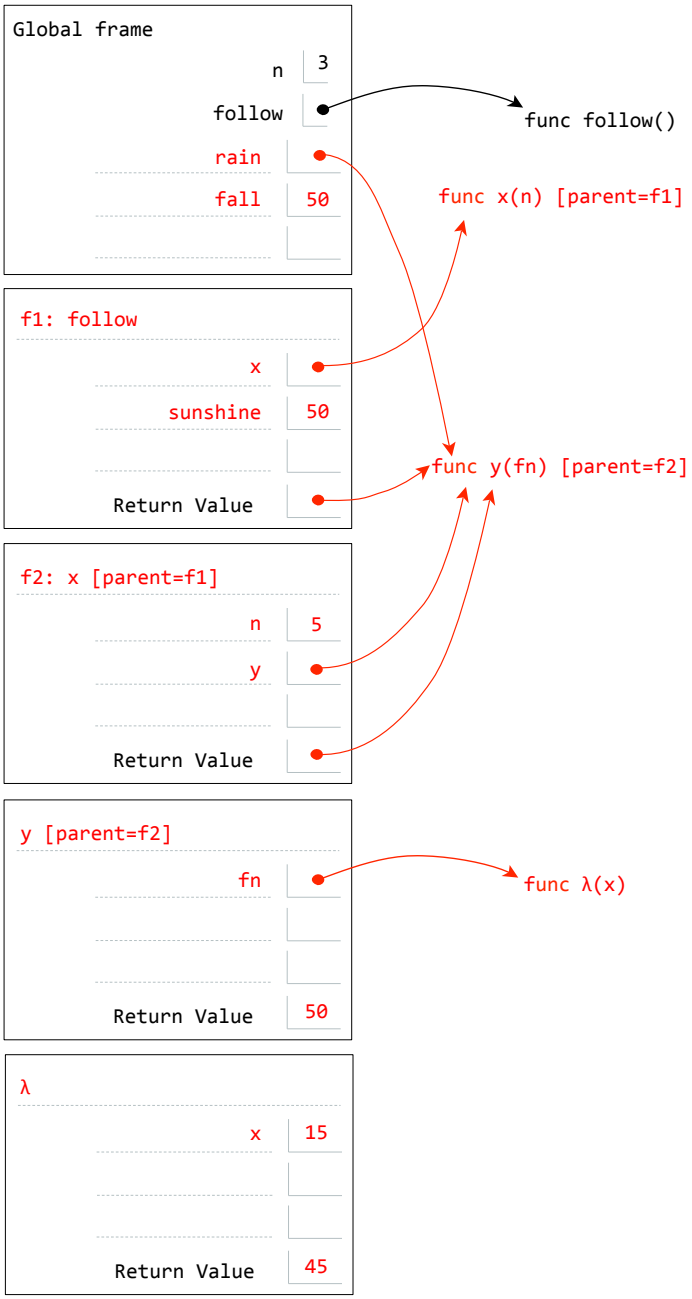
A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
n = 3

def follow():
    sunshine = 10
    def x(n):
        nonlocal sunshine
        def y(fn):
            nonlocal sunshine
            sunshine = fn(sunshine) + n
            return sunshine
        sunshine += n
        return y
    return x(5)

rain = follow()
fall = rain(lambda x: x * n)
```



4. (4 points) Good job, Mark

Mark is tired of having to dig around BearFacts to look up student information, so he wrote a script to create a dictionary of student info. However, when he did so, he was sleep-deprived. As such, he accidentally swapped the keys and the values! Now he's stuck with a dictionary that maps any single piece of student information to the student's login:

```
>>> info['Eric Tzeng'] # info is mark's faulty dictionary
'cs61a-ta'
>>> info['eric.tzeng@berkeley.edu']
'cs61a-ta'
```

Mark is too tired to fix this, so you'll have to do it for him. We'll do this by writing a `get_info` function. It should take a student login string and a faulty info dictionary as its two arguments and return a list containing all of that student's information. The returned list can contain a student's information in any order, but it must contain *all* of the information for a student found in the faulty dictionary.

```
>>> get_info('cs61a-ta', info)
['Eric Tzeng', 'eric.tzeng@berkeley.edu']
```

You may assume that no two students have duplicate information. However, you *may not* assume that names and emails are the only pieces of information stored in the dictionary.

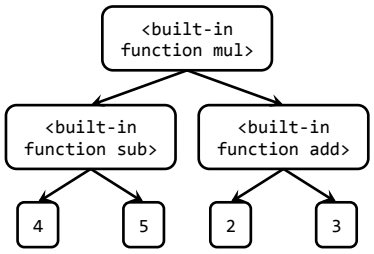
```
def get_info(login, info_dict):
    """Retrieve student information based on the student's login, using the
    provided (faulty) info_dict.

    >>> mark_dict = {'Andrew Huang': 'cs61a-tf', 'awesomeandrew@example.com': 'cs61a-tf',
    ...              'Robert Huang': 'cs61a-td', 'coolrob42@example.com': 'cs61a-td'}
    >>> get_info('cs61a-td', mark_dict)
    ['Robert Huang', 'coolrob42@example.com']  # order doesn't matter
    >>> get_info('cs61a-tf', mark_dict)
    ['awesomeandrew@example.com', 'Andrew Huang']  # order doesn't matter
    """
    info_list = []
    for info, account in info_dict.items():
        if account == login:
            info_list.append(info)
    return info_list
```

5. (5 points) Express yourself

In the first week of class, we learned that expressions like `mul(sub(4, 5), add(2, 3))` can be represented as expression trees. In this problem we use our `Tree` data structure to further explore this idea.

We propose representing expressions using a binary tree such as the `Tree` class you have seen in lecture. In this binary tree representation, leaves always represent atomic values such as `2` or `True`, and non-leaf nodes always represent function calls. For example, the expression `mul(sub(4, 5), add(2, 3))` corresponds to the following binary expression tree:



Write a function `evaluate`, which takes a binary expression `Tree` and returns what that expression would evaluate to. You may assume that all operators will take exactly 2 arguments, and that nodes always have either 0 or 2 children (never 1).

The class definition for `Tree` *can be found in the midterm study guide.*

```
def evaluate(exp):
    """Evaluates exp, which is an expression tree.

    >>> from operator import add, sub, mul
    >>> exp = Tree(3)  # 3
    >>> evaluate(exp)
    3
    >>> exp = Tree(add, Tree(3), Tree(4))  # add(3, 4)
    >>> evaluate(exp)
    7
    >>> exp = Tree(mul,
    ...            Tree(add, Tree(3), Tree(5)),
    ...            Tree(sub, Tree(5), Tree(2)))  # mul(add(3, 5), sub(5, 2))
    ...
    >>> evaluate(exp)
    24
    """
    if exp.left and exp.right:
        return exp.entry(evaluate(exp.left), evaluate(exp.right))
    return exp.entry
```

6. (8 points) The private life of Albert Wu

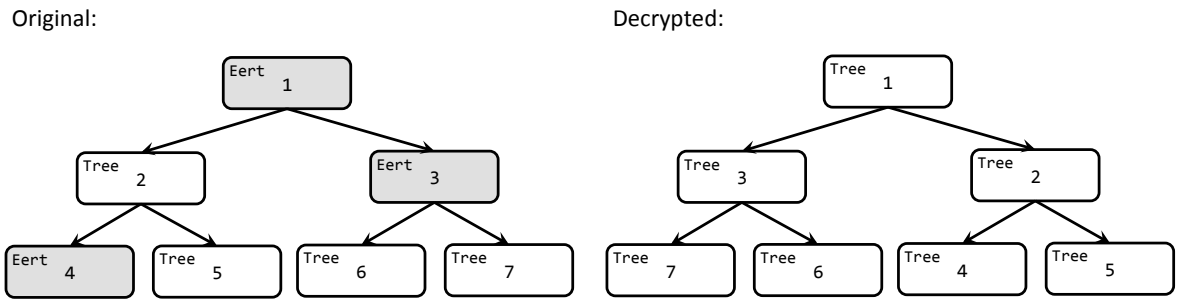
Albert keeps all of his top secret information in a binary tree. This prevents the layperson from reading his data. However, well trained computer scientists (such as you) can still access his information.

As a further layer of protection, Albert turns some of the nodes in his trees into `Eert` nodes. `Eert` nodes, which have `Tree` as their base class, are like normal `Tree` nodes, except they swap their left and right branches. (Albert settles for nothing less than the most advanced encryption techniques known to man.)

(a) (3 pt) Complete the `__init__` method for the `Eert` class **on the next page**. Make sure to use inheritance as much as possible. The `Eert` class should work as follows:

```
>>> e = Eert("61A account info",
...          Tree("Username: cs61a-te"),
...          Tree("Password: imsocool"))
>>> e.entry  # unchanged
"61A account info"
>>> e.left.entry  # swapped with right
"Password: imsocool"
>>> e.right.entry  # swapped with left
"Username: cs61a-te"
```

(b) (5 pt) Complete the definitions of the `decrypt` methods for both the `Tree` and `Eert` classes **on the next page**. When the `decrypt` method is invoked on a binary tree containing `Tree` and `Eert` nodes, it returns a copy of the binary tree, but with all `Eert` nodes replaced with `Tree` nodes. During this replacement, you should also swap the left and the right back to their proper positions! Here is a graphical representation of the process:



In terms of actual Python code, it should work as follows:

```
>>> old = Tree("League of Legends accounts",
...            Eert("Main account",  # swaps username and password positions
...                 Tree("Username: citrusdrink"),
...                 Tree("Password: imdiamond1")),
...            Tree("Smurf account",
...                 Tree("Username: yummycitrusdrink"),
...                 Eert("Password: ishouldbeapro")))  # no children, so no swapping
>>> new = old.decrypt()
>>> old.left.left.entry  # this got swapped because of the Eert
"Password: imdiamond1"
>>> new.left.left.entry  # no longer swapped
"Username: citrusdrink"
>>> old.right.right.entry
"Password: ishouldbeapro"
>>> new.right.right.entry
"Password: ishouldbeapro"
```

```
class Tree(object):

    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

    def decrypt(self):
        "*** PART B ***"
        left, right = None, None
        if self.left:
            left = self.left.decrypt()
        if self.right:
            right = self.right.decrypt()
        return Tree(self.entry, left, right)


class Eert(Tree):

    def __init__(self, entry, left=None, right=None):
        "*** PART A ***"
        Tree.__init__(self, entry, right, left)


    def decrypt(self):
        "*** PART B ***"
        tree = Tree.decrypt(self)
        tree.left, tree.right = tree.right, tree.left
        return tree
```

7. (6 points) kp it

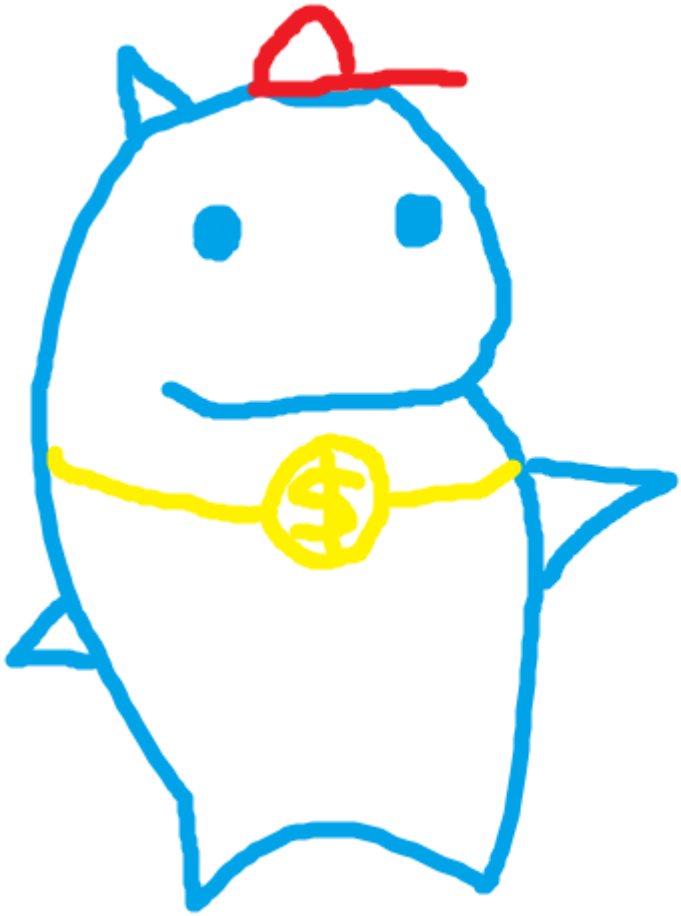
Complete the definition of the `skip` procedure *in Scheme*. The `skip` procedure should take a list as its only argument and return a copy of the list with every other element removed, beginning with the first element:

```
STk> (skip '(1 2 3 4 5))
(2 4)
STk> (skip '(a (b c) d e))
((b c) e)
STk> (skip '(50))
()
```

```
(define (skip lst)
  (if (<= (length lst) 1)
      '()
      (cons (cadr lst) (skip (cddr lst)))))
```

8. (0 points) Express yourself (v2)

Express your feelings in the space below through your choice of creative medium, such as poetry or illustration.



Comments

Problem 1

Grading: 1 pt each.

Problem 2

Grading: 2 pts each.

Common Mistakes: The first part looks very much like a linear-time function. However, since it can never recurse more than 23 times, it is actually constant time. (Can you see why?)

Problem 3

For both portions, the full rubric for the problem is available on Pandagrader.

Part A

Grading: 6 pts

Many students created extra list objects instead of removing and adding elements to the original list. This led to many students obtaining the incorrect final list, which should have been the reverse of the original. Some students also forgot that `pop` returns the object at the particular index and also removes it from the original list. It does not return the object inside a one-element list.

Many students also did not write that the return values for the `cup` frames are `None`. If a function does not explicitly have a return statement, it actually returns `None`.

For this problem, we established a baseline of 2 points. If the student created a global variable for `frosting` and bound it to a list, created one call frame for `cup`, and also bound the local `cake` variable to a list, the student could not receive lower than 2 points out of the maximum 6.

Part B

Grading: 6 pts

Many students did not understand how `nonlocal` affects the environment diagram and the outcome of the function calls. Students incorrectly created multiple `sunshine` variables and updated the newest one. Many students also thought that the lambda frames parent is the `y` frame. However, it is actually the global frame, and this affected the final outcome of `fall`.

Some students also did not realize that the `follow` and the `x` call frames should return the `y` function object. Many also did not bind `rain` to the `y` function object.

We established multiple baselines.

- Students received a minimum total of 1 point if they created a `follow` call frame, bound the `sunshine` variable to an integer and bound `x` to a function object.
- Students received a minimum total of 2 points if they met the 1 point baseline and also created a call frame for `x`, bound `n` to an integer, and bound `y` to a function object.
- Students received a minimum total of 3 points if they met the 2 point baseline and also had the correct return values for the `follow` and `x` call frames.
- Students received a minimum total of 5 points if they had everything correct, except used the `y` call frame as the parent of the lambda frame, and thus incorrectly obtained the final values for `fall`, `sunshine` and the return values.

Problem 4

Grading: 6 pts

Solutions were graded using the following rubric:

- 1 pt for iterating over the original dictionary in some way
- 1 pt for correctly retrieving values from the dictionary
- 1 pt for comparing values to login
- 1 pt for storing the keys of matching logins in a list

Common Mistakes: Most students iterated over the dictionary in some fashion, but quite a few did so incorrectly. Some students forgot that `for key in info_dict` only iterates over keys, and not the values as well. Other students forgot that the `keys`, `values`, and `items` attributes of dictionaries are methods, and thus didn't actually call the method properly.

As long as values were retrived from `info_dict` in some way, we awarded a point. This could be achieved through square-bracket notation (`info_dict[key]`) or by iterating over `info_dict.values()` or `info_dict.items()`.

A common mistake was comparing the keys to the login, rather than the value. A related error was storing values in the list, rather than the keys. Other students correctly stored the keys, but appended the values to the list incorrectly (e.g. `info += key`).

Problem 5

Grading: 5 pts

- 1 pt for checking if `exp` is a leaf
- 1 pt for returning `exp.entry` in the base case
- 1 pt for two recursive calls on the same code path, one on `exp.left` and one on `exp.right`
- 1 pt for attempting to use `exp.entry` as an operator
- 1 pt for correctly combining the recursive calls with the operator

Common Mistakes: For the base case, there were two common mistakes:

(a) Many people checked for `exp is None` rather than checking if `exp` was a leaf.

(b) Some people returned `exp`, which is a `Tree`, rather than `exp.entry`, which is the value stored in the `Tree`.

Some students checked the type of the entry, instead of checking whether or not it was a leaf. This was not accepted, since it is possible for the leaf entries to be functions.

Problem 6

Part A

- 1 pt for attempting to use inheritance by making some call to `Tree.__init__`
- 1 pt for correctly swapping the left and right children
- 1 pt for a functionally correct `Eert.__init__` method

Common Mistakes: The most common mistake by far was not using inheritance (i.e. not calling `Tree.__init__` at all). Many people also called the method, then manually swapped the branches afterward – but this is redundant, since we could've swapped the branches by passing them in the opposite order!

Part B

Grading: 5 pts

- 0.5 pt for correct base cases in `Tree.decrypt`
- 0.5 pt for correct base cases in `Eert.decrypt`
- 0.5 pt for returning a new `Tree` in `Tree.decrypt`
- 0.5 pt for returning a new `Tree` in `Eert.decrypt`
- 1 pt for making the correct recursive calls to `decrypt` in `Tree.decrypt`
- 1 pt for making the correct recursive calls to `decrypt` in `Eert.decrypt`
- 1 pt for swapping the left and right branches in `Eert.decrypt`

We graded solutions that involved calling `Tree.decrypt` from `Eert.decrypt` on a case-by-case basis. Errors like calling `decrypt` incorrectly or mutating the original `Tree`/`Eert` instance were penalized 0.5 pts.

Common Mistakes: The most common mistake was incorrectly calling the `decrypt` method. Some students didnt call the `decrypt` method and instead did something like `self.left.decrypt` (note the missing parentheses!). Others called `decrypt(self.left)`, which is an error because there is no `decrypt` function. Either of these errors resulted in a 0.5 pt penalty if the solution was otherwise correct.

The other most common mistake was not returning a new `Tree`. If the solution created a new `Tree` in all cases, the points for returning a new `Tree` were awarded. However, if this new tree was not returned, 0.5 pts were deducted.

Some other minor errors included:

- Forgetting to check whether the right or left branches were `None` before calling `decrypt` on them.
- Checking if `self == None` as a base case. This will never be `True` because `self` is an instance of a `Tree`, thus it can never be `None`.
- Checking if `self.entry` was a `Tree` or an `Eert` instead of self. This was solving a completely different problem!

Problem 7

Grading: 6 pts

- 1 pt for *a* base case
- 1 pt for *correct* base cases
- 1 pt for taking the `cadr` of `lst`
- 1 pt for taking the `cdr` of `lst` and recursing on it
- 1 pt for correctly recursing on the `cddr` of `lst`
- 1 pt for creating a new lst
- -1 pt for bad syntax
- -0.5 pt for miscellaneous errors

There are many ways to solve this problem. Other than the provided solution, you could also create a helper function:

```
(def (skip lst)
  (def (helper lst include?)
    (cond ((<= (length lst) 1) '())
          (include? (cons (car lst) (helper (cdr lst) #f)))
          (else (helper (cdr lst) #t))))
  (helper lst #f))
```

This helper solution keeps a boolean argument which alternates between `#t` and `#f`. It only keeps entries when the boolean is true.

Common Mistakes:

- Missing a base case (checking for the one element list)
- Failing to skip an element
- Scheme syntax errors, such as `(car (cdr (lst)))`

Problem 8

Many of you are very talented! Steven has provided a drawing of his own on these solutions.