CS 61A
Fall 2011

Structure and Interpretation of Computer Programs
Midterm Exam 2

INSTRUCTIONS

- You have 2 hours to complete the exam.

- The exam is closed book, closed notes, closed computer, closed calculator, except a one-page crib sheet of your own creation and the two official 61A midterm study guides.

- 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)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Total
/12	/10	/12	/16	/50

1. **(14 points)   Mutation and Non-Local Assignment**

The Berkeley Banking Commune only accepts Berkeley Bucks. Each valid "buck" has a unique serial number. The bank tracks these serial numbers to make sure that nobody is photocopying its bucks.

(a) **(4 pt)** The higher-order function `make_deposit` returns a single-argument function `deposit` that takes a Python list of serial numbers. If all serial numbers ever deposited to that function are unique, `deposit` returns the number of bucks ever deposited. If a duplicate serial number is ever deposited, `deposit` will forever return the string `'Fraud'`. Fill in the `deposit` definition below. Do not include any unnecessary `nonlocal` statements.
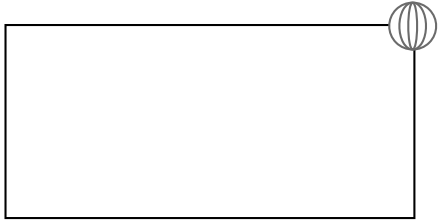
```python
def make_deposit():
    """Return a deposit function.

    >>> d1, d2 = make_deposit(), make_deposit()
    >>> d1([1, 3])
    2
    >>> d1([7, 7])
    'Fraud'
    >>> d1([8])
    'Fraud'
    >>> d2([2, 4, 6])
    3
    >>> d2([1, 3, 5])
    6
    >>> d2([4])
    'Fraud'
    """
    fraud = False
    contents = []
    def deposit(bucks):



            if fraud:
                return 'Fraud'
        return len(contents)
    return deposit
```

(b) (6 pt) Complete the environment diagram for the program in the box below. You **do not** need to draw an expression tree. A complete answer will:

- Complete all missing arrows. Arrows to the global frame can be abbreviated by small globes.
- Add all local frames created by applying user-defined functions.
- Add all missing names in frames.
- Add all **final** values referenced by frames. Represent tuple values using box-and-pointer notation.

```
big(b):
    if b == 0:
        return 2
    return 3 + big(b[0])
```

```
make_bank():
    bucks = 0
    def issue(n):
        nonlocal bucks
        bucks = (bucks, n)
        return big(bucks)
    return issue
```

```
issue(n):
    nonlocal bucks
    bucks = (bucks, n)
    return big(bucks)
```

```
def big(b):
    if b == 0:
        return 2
    return 3 + big(b[0])

def make_bank():
    bucks = 0
    def issue(n):
        nonlocal bucks
        bucks = (bucks, n)
        return big(bucks)
    return issue

s = make_bank()
s(1)
```

(c) (2 pt) After executing the program above, what would the expression s(4) evaluate to? If evaluating this expression causes an error, write "error." If evaluating this expression would take forever, write "infinite."

2. (10 points)   List Processing

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

```
from operator import add

def baby(oh):
    baby = oh
    for b in baby:
        baby[0] = b
    return baby

bieber = ['om', 'nom', 'nom']
counts = [1, 2, 3]
nums = counts
nums.append(4)
```

For each of the following expressions, write the `repr` string (i.e., the string printed by Python interactively) of the value *to which it evaluates* in the current environment . If evaluating the expression causes an error, write "error." Any changes made to the environment by the expressions below *will affect* the subsequent expressions.

(a) (2 pt) `baby(bieber)`

(b) (2 pt) `bieber[0:2]`

(c) (2 pt) `counts is nums`

(d) (2 pt) `counts is add([1, 2], [3, 4])`

(e) (2 pt) `tuple(map(baby, (nums, counts)))`

3. (12 points) Object-Oriented Programming

(a) **(2 pt)** Louis Reasoner wants to define a Person class:

```
class Person(object):
    name = None
    def __init__(self, name):
        Person.name = name
    def greet(self):
        return "Hello, my name is " + self.name
```

Alyssa P. Hacker, sees a problem. Circle **all** appropriate criticisms of this implementation.

(A) Every Person's name will be the equal to the most recently created Person's name.

(B) Instantiating a Person will cause an error.

(C) Every Person's name will be `None`.

(D) Invoking `greet` on a person instance will cause an error.

(b) **(2 pt)** Consider the following simple class definition.

```
class Dog(object):
    def bark(self):
        print("woof!")
```

One day, while using this class, Louis Reasoner decides he wants his dog, Fido, to `bark` differently:

```
>>> fido = Dog()
>>> fido.bark = "bow wow!"
```

Ben Bitdiddle quickly points out that this won't work. "`bark` is supposed to be a method, not a string!"
So Louis Reasoner attempts to reset the `bark` method to what it was before:

```
>>> fido.bark = Dog.bark
```

Ben replies, "I don't think your fix is right either!" Circle **all** appropriate statements about this final
assignment statement.

(A) Executing this assignment statement will cause an error.

(B) After this assignment, invoking `fido.bark()` will cause an error.

(C) This assignment statement will have no effect at all.

(D) None of the above criticisms are valid.

(c) (3 pt) Cross out statements below so that the expression `N().r()` evaluates to 1.

```python
class M(object):
    p = 2
    q = True
    def r(self):
        if self.q:
            return self.p
        return self.r() - 1

class N(M):
    p = 1
    q = False
    def r(self):
        return self.p + 1
```

(d) (2 pt) Consider the following class definition using our implemented object system (below-the-line).

```python
def make_foo_class():
    def __init__(self, a):
        self['set']('a', a)
    def someday(self):
        return "we will go " + str(attributes['a'])
    attributes = {'__init__': __init__,
                  'someday': someday,
                  'a': 42}
    return make_class(attributes, None)
Foo = make_foo_class()
```

Circle **all** equivalent class definitions below, among these options that use Python's built-in object system.

```python
class Foo:
    a = 42
    def __init__(self, a):
        self.a = a
    def someday(self):
        return "we will go " + str(Foo.a)
```

```python
class Foo:
    def __init__(self, a):
        self.a = a
    def someday(self):
        return "we will go " + str(self.a)
```

```python
class Foo:
    a = 42
    def __init__(self, a):
        self.a = a
    def someday(self):
        return "we will go " + str(a)
```

(e) (**3 pt**) Louis Reasoner decides to modify the seventh line of the body of `make_instance` in his implementation of an object system. He also implements a `Person` class.

```
 1: def make_instance(cls):
 2:     def get_value(name):
 3:         if name in attributes:
 4:             return attributes[name]
 5:         else:
 6:             value = cls['get'](name)
 7:             return value # CHANGED FROM: return bind_method(value, instance)
 8:     def set_value(name, value):
 9:         attributes[name] = value
10:     attributes = {}
11:     instance = {'get' : get_value, 'set' : set_value}
12:     return instance
13:
14: def make_person_class():
15:     def __init__(self, name):
16:         self['set']('name', name)
17:         Person['set']('population', Person['get']('population') + 1)
18:     def greet(self):
19:         return "Hello, my name is " + self['get']('name')
20:     Person = make_class({'__init__': __init__,
21:                          'greet': greet,
22:                          'population': 0})
23:     return Person
```

Circle any of the following lines that will cause an error using this implementation.


```
>>> Person = make_person_class()
```


```
>>> yakko = Person['new']('yakko')
```


```
>>> yakko['get']('name')
```


```
>>> yakko['get']('greet')()
```


```
>>> yakko['get']('population')
```

4. **(16 points) Defining Functions Without Iteration**

   ANSWER THESE QUESTIONS WITHOUT ANY FOR OR WHILE STATEMENTS

   (a) **(3 pt)** Implement the function `make_triangle_area` that defines a relation among the base `b`, height `h`, and area `a` of a triangle: $a = \frac{1}{2} \cdot b \cdot h$. You may assume that the functions `make_connector`, `multiplier`, and `constant` are defined for you. The following results should be printed using your solution.

   ```
   >>> a, b, h = [make_connector(n) for n in ('area', 'base', 'height')]
   >>> make_triangle_area(a, b, h)
   >>> a['set_val']('user', 75)
   area: 75
   >>> b['set_val']('user', 15)
   base: 15
   height: 10

   def make_triangle_area(a, b, h):
   ```

   (b) **(4 pt)** Write a function `overlap` that takes two words `word1` and `word2` and returns the maximum overlap between the end of `word1` and the beginning of `word2`. *Assume that both words have the same length.*

   ```
   >>> overlap('ball', 'ball')
   'ball'
   >>> overlap('pirate', 'teepee')
   'te'
   >>> overlap('fish', 'bowl')
   ''

   def overlap(word1, word2):
   ```

ANSWER THESE QUESTIONS WITHOUT ANY FOR OR WHILE STATEMENTS

(c) **(3 pt)** Complete the function `listify` that takes as an argument *either* a built-in `list` or a user-defined `Rlist` instance (defined in your study guide), and returns a `list` containing the same elements as the argument. Use type dispatching and a call to `listify_rlist`.

```
>>> r = Rlist('He will knock', Rlist(3, Rlist('times')))
>>> s = ["fezzes", "are", "cool"]
>>> listify(r)
["He will knock", 3, "times"]
>>> listify(s)
["fezzes", "are", "cool"]

def listify_rlist(r, s):
    """Fill list s with the contents of Rlist r."""
    if r is not Rlist.empty:
        s.append(r.first)
        listify_rlist(r.rest, s)

def listify(seq):
```
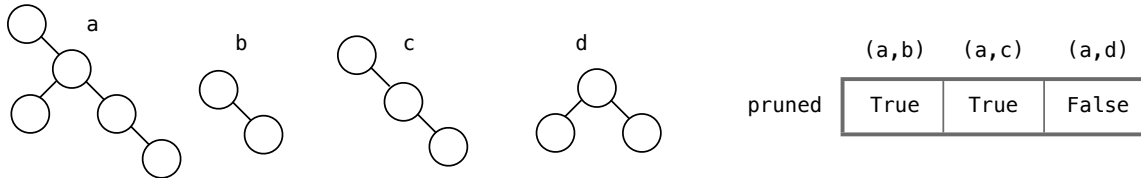
(d) **(2 pt)** Define a mathematical function $f(n)$ such that calling `listify(s)` on an `Rlist` instance `s` takes $\Theta(f(n))$ steps. Assume that `s` is an `Rlist` of length n and each element of `s` is itself an `Rlist` of length n. Assume that the `append` method for `list` takes a constant number of steps.

$f(n) =$

ANSWER THESE QUESTIONS WITHOUT ANY FOR OR WHILE STATEMENTS

Consider the binary `Tree` class below, which has no entry attribute.

```
class Tree(object):
    """A binary tree with no entries."""
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right
a = Tree(None, Tree(Tree(), Tree(None, Tree())))
b = Tree(None, Tree())
c = Tree(None, Tree(None, Tree()))
d = Tree(Tree(),  Tree())
```

	(a,b)	(a,c)	(a,d)
pruned	True	True	False

(e) (4 pt) Write a function `pruned` that takes two `Tree` arguments `t1` and `t2` and returns whether `t2` is a pruned version of `t1`. `t2` is a pruned version of `t1` if all paths from the root of `t2` are also valid paths from the root of `t1`. A path is an ordered sequence of branch selections (e.g., `right, right, left`). Your base case(s) should compare an input to `None`.

```
def pruned(t1, t2):
```