FINAL EXAMINATION

COMPUTER SCIENCE 61A

August 9, 2012

Instructions: Read Me!

- You will be given 180 minutes to work on this exam. **Do not start unless told to do so by the teaching staff.**
- This exam is closed book. Electronic devices (except dedicated timekeepers) must be turned off. You can use one double-sided 8.5” × 11” sheet of notes.
- Please write neatly and legibly, because *if we can’t read it, we can’t grade it*. If you are not sure of your answer, you may wish to provide a brief explanation.
- Finally, please take a deep breath and calm down before starting the exam. This exam is not worth having a heart attack for. We hope you do a CS61Awesome job!
- Thank you for being an awesome part of this class!

0 A Question of Identity (1 point)

Fill in the table below. **Once you are told to start, write your login at the top of each page**.

Name	
Login `cs61a-`	
All of the work on this exam is my own. (Please sign.)	

Question	0	1	2	3	4	5	6
Score	/1	/12	/6	/2	/6	/10	/6

7	8	9	10	11	12	**Total**
/5	/9	/5	/6	/6	/6	/80

1 Mind Your Language (12 points)

Face the Cons-equences

What would the STk Scheme interpreter print in response to the following lines of Scheme code?

(a) `STk> (cons (cons 1 2) (cons 3 (cons 4 5)))`

> **Solution:** `((1 . 2) 3 4 . 5)`

(b) `STk> (cdr (cons (list 1 2 3) (list 4 5 6)))`

> **Solution:** `(4 5 6)`

(c) `STk> (cdr (cdr (list 3 4 (list (cdr (list 6 7 8)) 9))))`

> **Solution:** `(((7 8) 9))`

Scheming a Calculated Approach

We would like to make a few changes to the interpreters for three languages: Calc, Py, and Scheme. Indicate whether the change should be made to the PARSER or the EVALUATOR of the interpreter. **You may only choose one of the two.**

(d) To make Py a lazy language, or a language that would only find the value of expressions when these values are needed, we would change:

PARSER **EVALUATOR**

(e) To allow Calc to understand *infix* operations, such as `5 + 2`, where the operator is between the operands that it operates on, we would change:

PARSER EVALUATOR

(f) To change the syntax of Scheme for our interpreter so that <> are used for Scheme lists and expressions instead of `()`, and a `|` is used instead of a `.` in pairs, we would change:

PARSER EVALUATOR

2 Jom Magrotker and the Order of the Growth (6 points)

(a)
```
def collide(n):
    lst = []
    for i in range(n):
        lst.append(i)
    if n <= 1:
        return 1
    if n <= 50:
        return collide(n - 1) + collide(n - 2)
    elif n > 50:
        return collide(50) + collide(49)
```

What is the order of growth in n of the runtime of `collide`, where n is its input?

$\Theta(1)$ $\Theta(\log n)$ [$\Theta(n)$] $\Theta(n \log n)$ $\Theta(n^2)$ $\Theta(2^n)$ $\Theta(n{\cdot}2^n)$ $\Theta(n^2{\cdot}2^n)$

(b)
```
def crash(n):
    if n < 1:
       return n
    return crash(n - 1) * n

def into_me(n):
    lst = []
    for i in range(n):
        lst.append(i)
    sum = 0
    for elem in lst:
        sum = sum + crash(n) + crash(n)
    return sum
```

What is the order of growth in n of the runtime of `into_me`, where n is its input?

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n \log n)$ [$\Theta(n^2)$] $\Theta(2^n)$ $\Theta(n{\cdot}2^n)$ $\Theta(n^2{\cdot}2^n)$

(c)
```
def carpe_diem(n):
    if n <= 1:
        return n
    return carpe_diem(n - 1) + carpe_diem(n - 2)
```

```
def yolo(n):
    if n <= 1:
        return 5
    sum = 0
    for i in range(n):
        sum += carpe_diem(n)
    return sum + yolo(n - 1)
```

What is the order of growth in n of the runtime of `yolo`, where n is its input?

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n \log n)$ $\Theta(n^2)$ $\Theta(2^n)$ $\Theta(n{\cdot}2^n)$ $\boxed{\Theta(n^2 \cdot 2^n)}$

3 Under Lock and Key-Value (2 points)

Suppose the following two statements are executed concurrently, in two separate threads, when `x` has a value of `5`:

```
x = x % 3
```

```
x = x + 20
```

The `%` operator returns the remainder when the first number is divided by the second.

(a) What are all of the possible values of `x` after the two statements are executed?

Solution: `22`, `1`, `2`, `25`

We now create a lock called `x_lock` and we modify the two threads from before:

```
x_lock.acquire()
x = x % 3
x_lock.release()
```

```
x_lock.acquire()
x = x + 20
x_lock.release()
```

As in the previous question, `x` has a value of `5`.

(b) What are now all of the possible values of `x` after the two threads have executed?

Solution: `22`, `1`

4 MapReduce: Not a Cartographer's Weight Loss Program (6 pts)

(a) Given a list of strings `wordlist`, fill in the blanks in the code below so that it returns a list of two elements, where the first element represents the number of strings with length greater than 3, and the second element is the number of strings of length 3 or less. For the list below, for example, the expected answer from the interpreter is `[2, 4]`.

```
>>> wordlist = ["it", "was", "the", "best", "of", "times"]
>>> reduce(lambda x, y: [x[0] + 1, x[1]] if y > 3 \
                        else [x[0], x[1] + 1],
           map(len, wordlist),
           [0, 0])
```

(b) Tom and Jon want to figure out who won the contest for project 4. They have a file, and each line of this file is a number that represents the entry that one student in the class votes for. Since the file is large, they want to use the MapReduce framework to determine *how many votes each entry has received*. They need a mapper and a reducer.

What should be the key and the value of each of the key-value pairs produced by the mapper, for each line in the file?

Solution: The key should be the entry number and the value should be `1`.

What should the reducer do with each key and the values associated with that key?

Solution: The reducer should sum up all of the values (the `1`s) associated with each key.

5 In Space, No One Can Hear You Stream (10 points)

(a) What are the first *five* values in the stream returned by the function given below?

```
def my_stream():
    def compute_rest():
        return add_streams(stream_filter(lambda x: x%2 == 0,
                                         my_stream()),
                           stream_map(lambda x: x+2, my_stream()))
    return Stream(2, compute_rest)
```

Solution: `2, 6, 14, 30, 62`

(b) Write a function `make_repeated_fn_stream` that, given a function `f` (of one argument), returns a stream such that the element at the ith location (counting from zero) is a *function* that would apply `f` i times to its argument. In other words, the stream would contain the elements

```
lambda x: x,
lambda x: f(x),
lambda x: f(f(x)),
lambda x: f(f(f(x))), ...
```

Note that these elements are functions, *not* strings.

The doctests for the function provide examples of the construction and usage of such a stream.

```
def make_repeated_fn_stream(fn):
    """Returns a stream where the ith element, counting from zero,
    is a function that applies fn i times.

    >>> square = lambda y: y * y
    >>> repeated_squares = make_repeated_fn_stream(square)
    >>> repeated_squares.first
    <function ...>
    >>> repeated_squares.first(3)
    3
    >>> repeated_squares.rest.first(3)
    9
    >>> repeated_squares.rest.rest.first(3)
    81
    """
```

Solution:

```
def compose(f, g):
    return lambda x: f(g(x))

def make_repeated_fn_stream(fn):
    def compute_rest():
        return stream_map(lambda g: compose(fn, g),
                          make_repeated_fn_stream(fn)))
    return Stream(lambda x: x, compute_rest)
```

Alternative solution:

```
def make_repeated_fn_stream(fn):
    def compute_rest():
        return stream_map(lambda g: lambda x: fn(g(x)),
                          make_repeated_fn_stream(fn)))
    return Stream(lambda x: x, compute_rest)
```

Alternative solution:

```
def repeated(fn, i):
    if i == 0:
        return lambda x: x
    else:
        return lambda x: fn(repeated(fn, i - 1)(x))

def make_repeated_fn_stream(fn):
    i = 1
    def compute_rest():
        nonlocal i
        repfn = repeated(fn, i)
        i += 1
        return Stream(repfn, compute_rest)
    return Stream(lambda x: x, compute_rest)
```

Alternative solution:

```
def make_repeated_fn_stream(fn):
    i = 1
    def compute_rest():
        nonlocal i
        repfn = lambda x: x
        for _ in range(i):
            repfn = compose(fn, repfn)
        i += 1
        return Stream(repfn, compute_rest)
    return Stream(lambda x: x, compute_rest)
```

6 Eeny, Meeny, Miny, Moe (6 points)

(a) Which of the following statements describes the purpose of the third message in a three-way handshake, which establishes a two-way connection between computer A and computer B? Computer A initiates the connection. Circle the option that is true.

1. For computer B to tell computer A that it can hear from computer A.
2. For computer A to tell computer B that it wants to start communicating with computer B.
3. For computer A to tell computer B that it can hear from computer B.

Comment: The order of the messages sent is 2, 1, 3.

(b) Which of the following statements about Newton's method, as discussed in lecture and lab, are true? Circle the options that are true. **All, some, or none of the statements may be true.**

1. Newton's method is guaranteed to find all the zeros of an equation.
2. Newton's method is guaranteed to find at least one zero of an equation.
3. Newton's method is guaranteed to find one zero of an equation.
4. Newton's method can find a zero of an equation if one exists, but is not guaranteed to.
5. Newton's method will always find the zero of an equation that is closest to the origin.
6. Newton's method will always find the zero of an equation that is closest to the initial guess.
7. Newton's method uses lazy evaluation to implement iterative improvement.
8. Newton's method can be used to write a function that finds the cube root of a number.

(c) Which of the following statements about the chat client and server, as discussed in lab and lecture, are true? Circle the options that are true. **All, some, or none may be true.**

1. To implement a broadcast message, both the client and server code need to be modified.
2. To implement a broadcast message, only the client code needs to be modified.
3. The three-way handshake can be a two-way handshake if the server sends multiple messages to the joining client, since the probability of the network dropping messages is reduced significantly.
4. To implement a broadcast message, each client needs to keep a copy of every client connected to the server to know whom to address each message to.

Comment: The client code would need to be modified to understand a new type of message from the user and send the appropriate message to the server. The server code would need to be modified to understand this new message from the user. The client would not need to keep a copy of every client connected to the server: it would merely send a message to the server, which maintains such a list. If the client kept such a list, it would not need to send a message to

the server! Also, it would be hard to synchronize this list between clients, as you will find out when you take a class in networking. Finally, even though the server sends a lot of messages, it is still possible that all of them would get lost – the probability is reduced, but not completely zero. The third message in the handshake allows the server to ensure that the client has indeed received the message from the server directed towards the client.

7 The Bro and Sis Code (5 points)

Write the function `are_siblings` that, given an ITree and two strings `s1` and `s2`, returns `True` if a node with datum `s1` and a node with datum `s2` are *siblings* in the tree. Two nodes are siblings if they have the same parent. You may assume that each string in the tree is unique: no name will occur more than once. **You may not convert the ITree to any other data structure.**

The doctests for the function provide examples of usage on the (truncated) Simpsons family tree.

```
def are_siblings(t, s1, s2):
    """Return true if a node with datum s1 is a sibling of a node
    with datum s2 in the ITree t.

    >>> simpsons_tree = make_itree('abraham',
    ...                            (make_itree('herb'),
    ...                             make_itree('homer',
    ...                                        (make_itree('lisa'),
    ...                                         make_itree('bart'),
    ...                                         make_itree('maggie')))))
    ...
    >>> are_siblings(simpsons_tree, 'lisa', 'bart')
    True
    >>> are_siblings(simpsons_tree, 'homer', 'bart')
    False
    >>> are_siblings(simpsons_tree, 'homer', 'moe')
    False
    """
```

Solution:

```
    if is_leaf(t):
        return False
    else:
        child_names = [itree_datum(c) for c in itree_children(t)]
        if s1 in child_names and s2 in child_names:
            return True
        else:
            for child in itree_children(t):
                if are_siblings(child, s1, s2):
                    return True
            return False
```

Alternative solution:

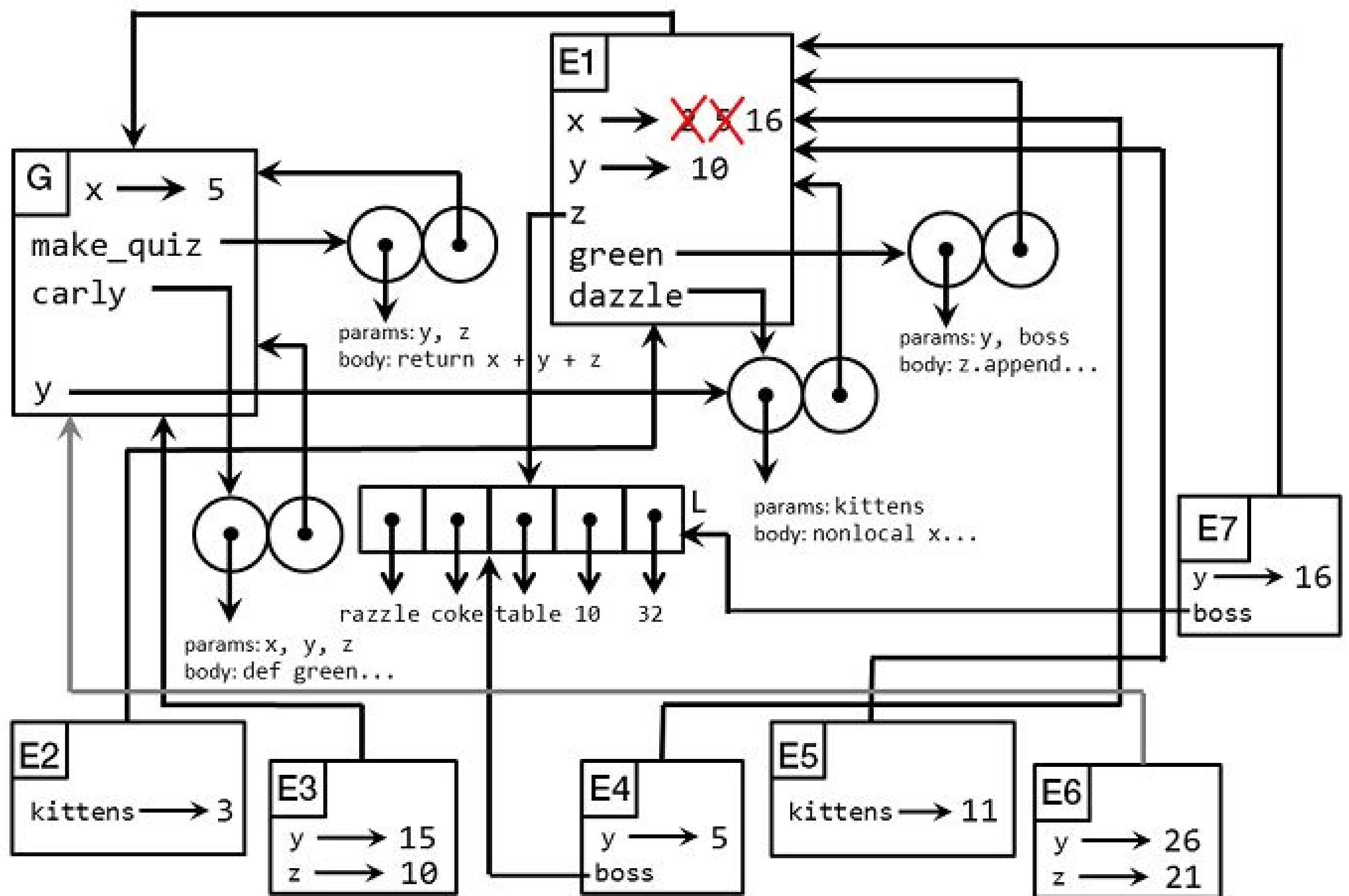
```
def are_siblings(t, s1, s2):
    if is_leaf(t):
        return False
    else:
        child_names = [itree_datum(c) for c in itree_children(t)]
        if s1 in child_names and s2 in child_names:
            return True
        return reduce(lambda x, y: x or y,
                      map(lambda child: \
                          are_siblings(child, s1, s2),
                          itree_children(t)))
```

8 C-x C-s the Environment (9 points)

Draw the environment diagram that results from the following interaction with the Python interpreter, and use the diagram to fill in the output of the last two function calls.

```
>>> x = 5
>>> def make_quiz(y, z):
...     return x + y + z
...
>>> def carly(x, y, z)
...    def green(y, boss):
...        z.append(x + y)
...        return 20
```

```
...     def dazzle(kittens):
...         nonlocal x
...         x += kittens
...         return make_quiz(x + y, x + 5) + green(x, z)
...     return dazzle
...
>>> y = carly(2, x + 5, ["razzle", "coke", "table"])
>>> y(3)
50
>>> y(11)
72
```



9 Iter Vegetables (5 points)

Provide the `__iter__` method for the simplified `RList` class given below, along with any necessary additional code, so that the doctests for the method would pass.

```
the_empty_rlist = None

class RList
    def __init__(self, first, rest=the_empty_rlist):
        self.first = first
        self.rest = rest

    def __iter__(self):
        """
        >>> my_rlist = RList(1, RList(2))
        >>> my_rlist_iter = iter(my_rlist)
        >>> next(my_rlist_iter)
        1
        >>> next(my_rlist_iter)
        2
        >>> next(my_rlist_iter)
        Traceback (most recent call last):
            ...
        StopIteration
        >>> my_rlist_iter2 = iter(my_rlist)
        >>> next(my_rlist_iter2)
        1
        """
```

Solution:

```
        current = self
        while current != the_empty_rlist:
            yield current.first
            current = current.rest
```

Alternative solution:

```
    def __iter__(self):
        self.current = self
        return self
```

```
    def __next__(self):
        if self.current == the_empty_rlist:
            raise StopIteration()
        result = self.current.first
        self.current = self.current.rest
        return result
```

10 Don't Repeat Yourself (6 points)

For this problem, we will use the *unary representation* of numbers as described in homework, where a number is represented as a list of the same number of `a`'s. So, for example, the number 3 is represented as `<a, a, a>` and the number 0 is represented as `<>`. Write the rules and associated facts for `repeat`, which relates three lists in the following manner:

```
P?> repeat(<foo, bar>, <<a, a, a>, <a, a>>, <foo, foo, foo, bar, bar>)
Yes.
P?> repeat(<foo, bar, garply>, <<a, a>, <>, <a, a, a>>, ?what)
Yes.
?what = <foo, foo, garply, garply, garply>
P?> repeat(<foo, bar>, ?what, <foo, bar, bar, bar, bar>)
Yes.
?what = <<a>, <a, a, a, a>>
P?> repeat(<foo, bar>, ?what, <bar, foo, foo>)
No.
```

The ith element of the first list is repeated in the third list as many times as specified by the ith element of the second list. So, in the first example, the first element of the first list (`foo`) is repeated in the third list as many times as specified by the first element of the second list (`<a, a, a>`).

Solution:

```
fact repeat(<>, <>, <>)
rule repeat(<?f | ?r>, <<?num_f | ?num_r> | ?other_nums>,
            <?f | ?result_r>):
    repeat(<?f | ?r>, <?num_r | ?other_nums>, ?result_r)

rule repeat(<?f | ?r>, <<> | ?other_nums>, ?result_r):
    repeat(?r, ?other_nums, ?result_r)
```

Alternate solution:

```
fact repeat_one(?f, <>, <>)
rule repeat_one(?f, <a | ?num_r>, <?f | ?result_r>):
    repeat_one(?f, ?num_r, ?result_r)

fact repeat(<>, <>, <>)
rule repeat(<?f | ?r>, <?num_f | ?num_r>, ?result):
    repeat_one(?f, ?num_f, ?result_f)
    repeat(?r, ?num_r, ?result_r)
    append(?result_f, ?result_r, ?result)
```

11 I Can Haz Cycle? (6 points)

We will call a dictionary *cyclic* if there is at least one sequence of key-value pairs such that the value of one pair matches the key of the next pair, and the value of the last pair matches the key of the first pair. For instance, consider the following dictionary:

```
example_d = { 'awesome': 'bodacious', 'soda': 'cs',
              'bodacious': 'soda', 'cs': 'awesome', '61a': 'cool' }
```

This dictionary is cyclic because we have the chain:

```
'awesome' -> 'bodacious' -> 'soda' -> 'cs' -> 'awesome'
```

where `awesome` maps to `bodacious`, `bodacious` maps to `soda`, and so on, until we find `awesome` again. Note that the entire dictionary does **not** need to be a cycle. In the dictionary `example_d`, for instance, there is a key-value pair that is not part of the cycle: `('61a', 'cool')`. The dictionary is still considered cyclic because at least one cycle exists.

Define the procedure `is_cyclic` that, given a dictionary `dict`, returns `True` if `dict` is cyclic.

Hint: It may be helpful to write a helper that takes a starting key and checks if there is a cycle starting with that key.

```
def is_cyclic(dict):
```

Solution:

```
    def cycle_helper(d, k):
        """Checks if there is a cycle in d
           that starts with key k."""
```

```
        history = []
        while k in dict:
            v = dict[k]
            if v in history:
                return True
            history.append(v)
            k = v
        return False

    for k in dict:
        if cycle_helper(dict, k):
            return True
    return False
```

12 See OOP Run (6 points)

Consider the following definition, in the Python-native OOP system, of the class `PetDog`, which inherits from the `Animal` class (not shown).

```
class PetDog(Animal):
    num_petdogs = 0

    def __init__(self, name, owner):
        Animal.__init__(self, name)
        self.owner = owner
        PetDog.num_petdogs += 1

    def eat(self, food):
        Animal.eat(self, food)
        print("BARK BARK BARK")
        self.sleep()

    def sleep(self):
        print(self.owner)
```

Assuming that `make_animal_class` was defined already, complete the code on the next page so that we would have an equivalent `PetDog` class that uses the object-oriented programming implementation from lecture, lab and discussion, which uses dispatch dictionaries.

Note: Do not forget to assign to the variable `PetDog`!

```
Animal = make_animal_class()

def make_petdog_class():
    """Creates a PetDog class."""
```

Solution:

```
    def __init__(self, name, owner):
        Animal['get']('__init__')(self, name)
        self['set']('owner', owner)
        PetDog['set']('num_petdogs',
                      1 + PetDog['get']('num_petdogs'))

    def eat(self, food):
        Animal['get']('eat')(self, food)
        print("BARK BARK BARK")
        self['get']('sleep')()

    def sleep(self):
        print(self['get']('owner'))

    attributes = {'__init__': __init__,
                  'eat': eat,
                  'sleep': sleep,
                  'num_petdogs': 0}

    return make_class(attributes, Animal)

PetDog = make_petdog_class()
```