Midterm Examination 1

Computer Science 61A

July 9, 2012

Instructions: Read Me!

- You will be given 105 minutes to work on the individual portion of the midterm, and 15 minutes to work on the group portion. Please 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" $\times$ 11" sheet of handwritten 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!

0 A Question of Identity (1 point)

Write your name and login **at the top of each page of the midterm**. Also, fill in the table below:

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

Question	0	1	2	3	4	5	6	**Total**
Score	/1	/9	/8	/6	/12	/7	/7	/50

1 So Call Me, Maybe (9 points)

For each of the following call expressions, write the value to which it evaluates (which may differ from what is printed) and what would be printed at the interpreter.

In the column labeled **Evaluates to**, write the value the call expression evaluates to. If it produces an error, write "ERROR" and describe the reason for the error.

In the column labeled **Prints at Interpreter**, write what would be printed during an interactive session of the Python interpreter. You need to include everything that is printed: if there are multiple lines printed, print them in the same order as the Python interpreter would. If the interpreter produces an error, also describe the reason for the error. If there are multiple possible errors, describe only the first one that the Python interpreter would reveal.

Assume that we have typed the following code at the Python interpreter:

```python
from operator import mul
def square(x):
    return mul(x, x)
def print_square(x):
    print(mul(x, x))
```

As an example, we show you what the first two expressions evaluate to, and what would be printed.

Expression	Evaluates to	Prints at Interpreter
`square(4/0)`	Error, Division by zero	Error, Division by zero
`square(2)`	4	4
`print(5)`		
`square(add(square(2), 2))`		
`print_square(add(square(2), 2))`		
`square(print_square(2))`		
`print(square(4))`		
`square(print(4))`		

2 Potpourri (8 points)

(a)
```
def lucky():
    def charms():
        print('citrus')
    print('drink')
    return charms
```

Given the above definition, what would the Python interpreter print in response to the following expression call?

```
>>> lucky()()
```

(b)
```
bar = (lambda x, y: y)((lambda z: (z, 2))(print(1)))
```

Given the above definition, what is the value of `bar`?

(c)
```
from operator import add
from functools import reduce
foo = reduce(add,
             filter(lambda x: x > 5,
                    map(lambda tup: tup[tup[0]],
                        ((1, 1, 2), (2, 3, 6),
                         (1, 30, 40), (0, 50, 50)))))
```

Given the above definition, what is the value of `foo`?

(d) Draw the box-and-pointer diagram for the IRList `<1, <2, 3>, 4>`.

3 Growing Pains (6 points)

As part of a program for communications research, Cecilia writes the following piece of code:

```
def sum_to_n(n):
    if n == 0:
        return 0
    else:
        return n + sum_to_n(n - 1)

def compute_result(n):
    i = 0
    result = 0
    while i < n:
        result += sum_to_n(n)
        i += 1
    return result
```

(a) Cecilia (correctly) determines that the order of growth of `compute_result` in time is in $O(n^2)$, where n is the input to `compute_result`. While debugging her program, she makes the following change to her program, where the change is shown with a comment:

```
def sum_to_n(n):
    if n == 0:
        return 0
    else:
        return n + sum_to_n(n - 1)

def compute_result(n):
    i = 0
    result = 0
    while i < n:
        result += sum_to_n(i)           # n changed to i.
        i += 1
    return result
```

What is the new order of growth of `compute_result` in time, for any input n? Circle the most appropriate answer from the options below:

$O(1)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$ $O(n \cdot 2^n)$ $O(n^2 \cdot 2^n)$

(b) She changes the `i` back to an `n`. However, she also modifies `sum_to_n`, where the change is shown with a comment:

```
def sum_to_n(n):
    if n == 0:
        return 0
    else:
        # Line below changed.
        return n + sum_to_n(n - 1) + sum_to_n(n - 1)

def compute_result(n):
    i = 0
    result = 0
    while i < n:
        result += sum_to_n(n)          # i changed back to n.
        i += 1
    return result
```

What is the new order of growth of `compute_result` in time, for any input n? Circle the most appropriate answer from the options below:

$O(1)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$ $O(n \cdot 2^n)$ $O(n^2 \cdot 2^n)$

(c) Elsewhere in the program, Cecilia has written the following piece of code:

```
def biz(n):
    if n < 0:
        return 50
    return biz(n - 2) + biz(n - 1)

def fooply(n):
    if n < 0:
        return 1
    return biz(5000) + fooply(n - 1)
```

What is the order of growth of `fooply` in time, for any input n? Circle the most appropriate answer from the options below:

$O(1)$ $O(n)$ $O(n \log n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$ $O(n \cdot 2^n)$ $O(n^2 \cdot 2^n)$

4 Song for the Broken Abstractions (12 points)

As part of a program to understand music trends, we would like to create abstract data types (ADTs) for songs and albums. Here is the definition for the constructors and selectors of an abstract data type representing songs:

```
def make_song(name, bpm, likes):
    return (name, bpm, likes)

def get_song_name(song):
    return song[0]
def get_bpm(song):
    return song[1]
def get_likes(song):
    return song[2]
```

`bpm` refers to the number of beats per minute (BPM) in the song: a higher BPM implies a faster song. `likes` refers to the number of people who have liked the song, as obtained from surveys.

(a) We will also create the ADT for albums. An album has a name and a collection of songs, represented as a tuple. This is the constructor we will use:

```
def make_album(name, songs):
    return make_irlist(name, make_irlist(songs))
```

Write the selector functions `get_album_name` and `get_songs` that return the name and the tuple of songs, respectively.

```
def get_album_name(album):


def get_songs(album):
```

(b) We have now written a sizable amount of code involving albums and songs. If we modify the definition of the constructor `make_album` to instead be:

```
def make_album(name, songs):
    return (name, songs)
```

What other functions will we have to change to ensure that code that depends on the album ADT does not break? Circle those functions among the options provided below.

`get_album_name`   `get_songs`   `get_song_name`   `get_bpm`   `get_likes`

(c) Louis Reasoner, a fellow programmer on the project, would like to use our ADTs to write a utility function that creates an album of popular songs from a given album. A song is popular if it has more than a certain number of likes.

However, he has a few data abstraction violations in his code. For lines in his code that have violations, scratch out the line and rewrite it in the blank space *just below* with a fix that removes the violation. Not all lines have violations.

Note: Assume the *new* definition of the album ADT *from part (b)*, **not** from part (a).

```
def popular_songs_album(album, new_name, like_limit=50):

    popular_songs = ()

    for song in album[1]:

        if song[2] >= like_limit:

            popular_songs = popular_songs + (song,)

    return (new_name, popular_songs)
```

(d) Now, using the ADTs we created above, write the function `get_avg_bpm`, which returns the average BPM across all songs on the album provided as an argument. Assume that the album contains at least one song. Remember to respect the data abstraction!

```
def get_avg_bpm(album):
```

5 The Search for Truth (7 points)

We have a sequence of numeric data points `seq1`, and we want to see if a sequence of relevant numbers `seq2` is found in the data. The catch is, we want to see if the numbers in `seq2` occur *in the same order* within the data of `seq1`, though not necessarily one after the other. If so, then `seq2` is a *subsequence* of `seq1`.

Write a predicate function `is_subseq` that takes two tuples `seq1` and `seq2` as arguments, and determines if `seq2` is a subsequence of `seq1`. If so, the function should return `True`; otherwise, it should return `False`. We have provided a few doctests to demonstrate the definition and usage.

Your solution should use recursion. Do not use a `while`-loop or `for`-loop in your solution.

```
def is_subseq(seq1, seq2):
    """Returns True if seq2 is a subsequence of seq1.

    >>> is_subseq((9, 1, 4, 5, 6), (4, 5, 6))
    True
    >>> is_subseq((3, 5, 0, 3, 4, 3, 7, 9, 3, 2), (3, 3, 9, 2))
    True
    >>> # Below, the numbers in seq2 appear in seq1,
    >>> # but not in the same order.
    >>> is_subseq((3, 5, 5, 8, 3), (8, 5, 3))
    False
    >>> # Below, not all the numbers in seq2 are present in seq1.
    >>> is_subseq((3, 5, 5, 8, 3), (3, 2, 8))
    False
    >>> is_subseq((3, 2, 57, 8), (3, 5, 7))
    False
    """
```

6 Counting in the Deep (7 points)

Write the function `count_occur` that takes an IRList and returns the number of times a certain element appears in the IRList. The IRList can be arbitrarily deep. We have provided a few doctests to demonstrate usage. You can continue your solution on the next page, if you need to, but **make it clear to us that you are continuing on the next page.**

```
def count_occur(deep_irlist, element):
    """Returns the number of times element occurs in deep_irlist.

    >>> test_irlist1 = irlist_populate(1, 2, 3)
    >>> irlist_str(test_irlist1)
    '<1, 2, 3>'
    >>> test_irlist2 = irlist_populate(irlist_populate(3, 2, 6),
                                       make_irlist(make_irlist(6)))
    >>> irlist_str(test_irlist2)
    '<<3, 2, 6>, <<6>>>'
    >>> count_occur(test_irlist2, 6)
    2
    >>> test_irlist_deep = irlist_populate(test_irlist1, test_irlist2)
    >>> irlist_str(test_irlist_deep)
    '<<1, 2, 3>, <<3, 2, 6>, <<6>>>>'
    >>> count_occur(test_irlist_deep, 3)
    2
    >>> count_occur(test_irlist_deep, 8)
    0
    """
```

```
# Continue your solution for count_occur here,
# if you need to.
```