MIDTERM EXAMINATION 1: SOLUTIONS

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.

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

```
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)`	`None`	`4`
`square(add(square(2), 2))`	`36`	`36`
`print_square(add(square(2), 2))`	`None`	`36`
`square(print_square(2))`	`Error`: Cannot multiply `None`	`4`, followed by `Error`: Cannot multiply `None`
`print(square(4))`	`None`	`16`
`square(print(4))`	`Error`: Cannot multiply `None`	`4`, followed by `Error`: Cannot multiply `None`

Comments: The answers in the first two rows were worth a quarter of a point each, while the rest were worth 1 full point. All answers were graded or nothing. Many students correctly indicated that `square(print_square(2))` and `square(print(4))` would print an error, but forgot to indicate that these lines also printed `4` before printing the error.

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()()
```

Solution:

```
drink
citrus
```

This question tested the order of evaluation of operator and operands. In the first week, we mentioned that the operator is evaluated first, followed by the operands from left to right, and *then* we call the operator on the operands. The operator of the outermost function call is the expression `lucky()`, while there are no operands. Now, when evaluating `lucky()`, we have to (again) evaluate the operator and then the operands, of which there are none. The operator of *this* expression is `lucky`, which evaluates to a function: this function is named `charms` in the body of `lucky`. During evaluation, however, the string `drink` is printed.

When we finally call the result of evaluating `lucky()` in the expression `lucky()()`, after having evaluated the operator and the operands, we evaluate the body of `charms`, which prints the string `citrus`.

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

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

Solution: This results in an error. Again, the operator is evaluated first, which evaluates to an anonymous function that expects two arguments. Then, the (only) operand is evaluated, which itself turns out to be calling an anonymous function with one argument (`z`) on one operand (`print(1)`). The operand evaluates to `None`. (Remember the difference between evaluation and printing! This is one of the main ideas this, and the previous, question was testing.) The anonymous function with one argument is called on `None`, and returns the tuple `(None, 2)`. The outermost anonymous function expects two arguments, but it

only receives *one*, which is a tuple. Even though the tuple has two elements, it is treated as one unit.

If we wanted to use both elements of the tuple separately, we would use the `*` symbol, which "expands" the tuple to fit the arguments of the outermost anonymous function. A better statement that may have fit the programmer's intent would have been

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

The staff decided that the question was too tricky, and took only a half point off if you assumed that the function call correctly occurred with two arguments and thus concluded that the value was `2`.

(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`?

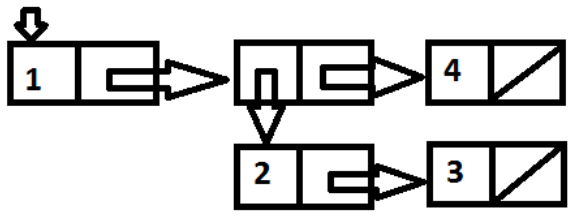
Solution: `36`

We start from the inside out. The call to `map` extracts the element of each tuple that was specified by the first element of each tuple. That's what `tup[tup[0]]` was trying to accomplish: it grabbed the first element of each tuple and then used that element to index back into the *same* tuple.

Once we have this sequence of numbers (`1`, `6`, `30`, and `0`), we use `filter` to remove those numbers that are not greater than 5. We now have the sequence of numbers `6` and `30`, which we then reduce using the `add` operator to get `36`.

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

Solution:



One way to approach box-and-pointer diagrams like this is to recognize that an IRList is merely a collection of nested pairs. This makes them incredibly useful and recursive, since you can peel off the outermost pair and still be left with another IRList. If there are n elements in an IRList, there are n nested pairs, and the left half of each pair is a particular element in the IRList. Here, the outermost IRList had three elements, so there are three nested pairs. The left half of each pair contains an element in the IRList.

It just so happens that the second element is *also* an IRList that has two elements. Following the same rules above, the second element is represented by two nested pairs, the left halves of which are the elements in the nested IRList.

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:

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

Solution: The piece of code in `compute_result` that proves important in finding the order of growth in time is the loop. The first iteration of the loop calls `sum_to_n` with an argument of 0, the second iteration calls `sum_to_n` with an argument of 1, and so on. The total amount of time taken by `sum_to_n` is therefore proportional to $0+1+2+\ldots+(n-1)$, which is in $O(n^2)$.

(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:

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

Solution: The function `sum_to_n` now has a running time in $O(2^n)$. This is because each call to `sum_to_n` produces (in the worst case) two more calls to `sum_to_n`, resulting in (at most) 2^n calls to `sum_to_n`. Since this function is called n times in `sum_to_n`, the total running time is in $O(n \cdot 2^n)$.

Notice that if Cecilia had replaced the two recursive calls to `sum_to_n` with

```
2 * sum_to_n(n - 1),
```

then the running time drastically reduces to $O(n^2)$. This is a great (if extreme) example of how a knowledge of the problem and some of the mathematics behind it can significantly reduce the running time of the program, allowing it to deal with much larger inputs quicker.

(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:

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

Solution: The function `fooply` recursively calls itself n times, and each time it does a constant amount of work to calculate `biz(5000)`. Notice that this work takes a constant amount of time each time – even though the constant is huge ($\approx 2^{5000}$) – since the amount of time taken to calculate `biz(5000)` does not depend on 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):
```

Solution:

```
    return irlist_first(album)
```

```
def get_songs(album):
```

Solution:

```
    return irlist_first(irlist_rest(album))
```

Many students forgot to call `irlist_first`. Remember that `irlist_rest` returns an IRList. We wanted the tuple of songs that was the second element of the list. Another valid answer was

```
    return irlist_select(album, 1)
```

(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`

Comments: Some students chose not to circle `get_album_name`. While `get_album_name` would technically still work due to the underlying implementation of IRLists, it would be a data abstraction violation to not change *how* the elements are selected.

(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 get_songs(album):                # Corrected.

        if get_likes(song) >= like_limit:        # Corrected.

            popular_songs = popular_songs + (song,)

    return make_album(new_name, popular_songs)   # Corrected.
```

Comments. A common mistake was to not call `make_album` in the **`return`** statement.

(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):
```

Solution:

```
average = 0
songs = get_songs(album)
for song in songs:
    average += get_bpm(song)
return average / len(songs)
```

Comments. A common mistake was treating `album` as if it were a tuple, rather than using the provided selectors. This was a data abstraction violation.

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
    """
```

Solution:

```
    if seq2 == ():
        return True
    elif seq1 == ():
        return False
    elif seq1[0] == seq2[0]:
        return is_subseq(seq1[1:], seq2[1:])
    else:
        return is_subseq(seq1[1:], seq2)
```

The idea behind the solution was to progressively examine the first element of `seq1` and check if it is the first element of `seq2`. If it is, then we have accounted for one element of `seq2`, and we can ignore it for future calls to `is_subseq`.

For this solution, the order of the base cases was important. If `seq2` is empty, this implies that there are no elements in `seq2` to check for existence in `seq1`, so trivially, `seq2` *is* a subsequence of `seq1`. However, if `seq2` is not empty but `seq1` is, then `seq1` can no longer possibly have elements in `seq2`, so `seq2` cannot be a subsequence of `seq1`.

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
"""
```

Solution:

```
if deep_irlist == empty_irlist:
    return 0
if could_be_irlist(irlist_first(deep_irlist)):
    return count_occur(irlist_first(deep_irlist), element) + \
           count_occur(irlist_rest(deep_irlist), element)
if irlist_first(deep_irlist) == element:
    return 1 + count_occur(irlist_rest(deep_irlist), element)
return count_occur(irlist_rest(deep_irlist), element)
```

Common mistakes were forgetting to recurse into a "deep" first element (where the first element was also an IRList), breaking abstraction barriers for IRlists (such as trying to index in the `deep_irlist` or using a **`for`**-loop), or solving the problem as if IRLists were tuples.