CS 61A Structure and Interpretation of Computer Programs

Summer 2013 Midterm 1 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)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6	Total
/12	/5	/3	/11	/7	/12	/50

1. (12 points) Proceed with call-tion

For each of the following expressions, write the value to which it evaluates *and* what would be output by 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:

```
from operator import mul

x = 3

def square(x):
    return mul(x, mul(x, 1))

def blaster(y):
    return print(square(y) + x)
```

Expression	Evaluates to	Interactive Output
`square(7)`	49	49
`1/0`	Error	Error
`square(2) + square(x)`	13	13
`print(square(3))`	None	9
`blaster(5)`	None	28
`print(blaster(2) + 5)`	Error	7 Error
`blaster(blaster(3))`	Error	12 Error
`25 or (5 / 0)`	25	25

2. (5 points) Lambda? No thanks, I prefer chicken

(a) (2 pt) Fill in the blanks below so that `foo(5)(10)()` returns `[5, 10]`. You may *not* write any numbers in your solution, and you may only add expressions in the blanks.

```
foo = lambda __________: lambda y: _____________________________________________
```

```
foo = lambda x: lambda y: lambda: [x, y]
```

(b) (3 pt) Fill in the blanks below so that the final call expression below evaluates to a *tuple* value. For this section, you *may* write numbers, but not tuples, and you may only add expressions in the blanks.

```
def love(x):
    if x == 'zedd':
        return [1, 2, lambda: (2, 3)]
    else:
        return lambda: 5

(lambda ______________, banana: foxes_______________________________)(love, 'clarity')
```

```
(lambda foxes, banana: foxes('zedd')[2]())(love, 'clarity')
```

3. (3 points) Tracing through the facts

Consider the following portion of code:

```
def tracer(fn):
    def traced(x):
        print('Calling', fn, '(', x, ')')
        result = fn(x)
        print('Got', result, 'from', fn, '(', x, ')')
        return result
    return traced

def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)

new_fact = tracer(fact)
```

Circle the **Choice X** heading of one of the options below corresponding to what Python would display if we ran `new_fact(2)` in an interpreter session. You may assume that the "ADDRESS" in each output is correct.

<table>
<tr>
<td>Choice A
Calling <function fact at ADDRESS> (2)
Calling <function fact at ADDRESS> (1)
Calling <function fact at ADDRESS> (0)
Got 1 from <function fact at ADDRESS> (0)
Got 1 from <function fact at ADDRESS> (1)
Got 2 from <function fact at ADDRESS> (2)
2</td>
<td>Choice B
Calling <function fact at ADDRESS> (2)
Got 2 from <function fact at ADDRESS> (2)
Calling <function fact at ADDRESS> (1)
Got 1 from <function fact at ADDRESS> (1)
Calling <function fact at ADDRESS> (0)
Got 1 from <function fact at ADDRESS> (0)
2</td>
</tr>
<tr>
<td>Choice C
Calling <function fact at ADDRESS> (2)
Got 2 from <function fact at ADDRESS> (2)
2</td>
<td>Choice D
2</td>
</tr>
<tr>
<td colspan="2">Choice E
The output is none of the above. If you select this choice, please briefly explain why:</td>
</tr>
</table>

4. (11 points) Save the environment (diagrams)!

(a) (5 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.*

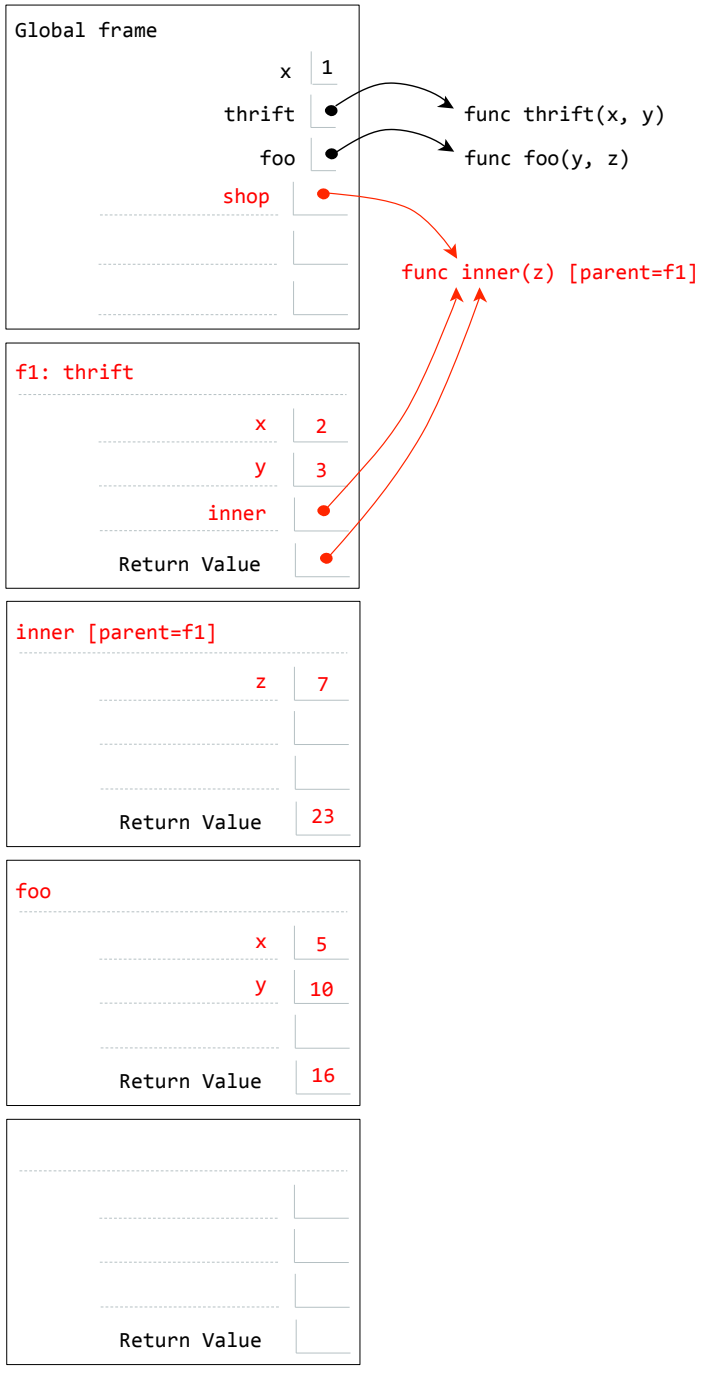
A complete answer will:

- 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.

```
x = 1
def thrift(x, y):
    def inner(z):
        return foo(5, 10) + z
    return inner

def foo(y, z):
    return x + y + z

shop = thrift(2, 3)
shop(7)
```



(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.*

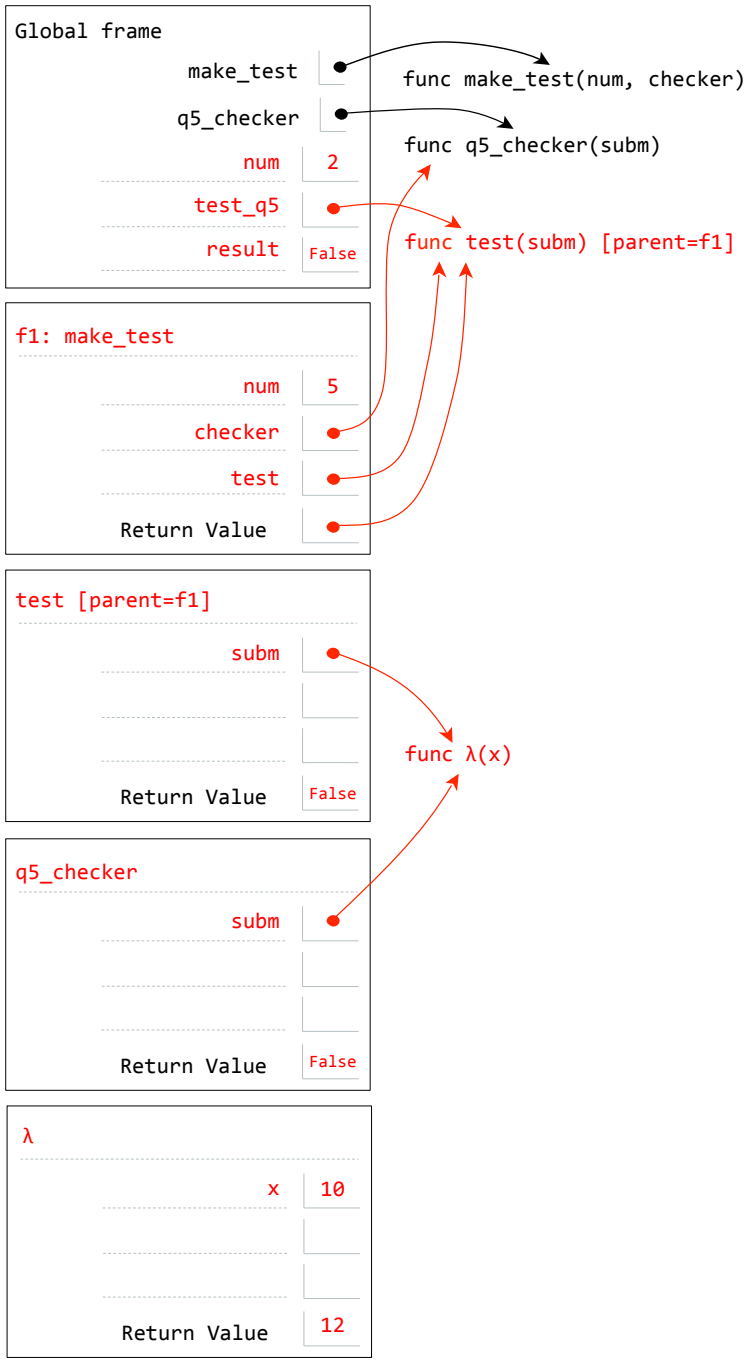
A complete answer will:

- 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.

```
def make_test(num, checker):
    def test(subm):
        return checker(subm)
    return test

def q5_checker(subm):
    return subm(10) == 15

num = 2
test_q5 = make_test(5, q5_checker)
result = test_q5(lambda x: x + num)
```



5. (7 points) Testing our (pot)luck

While planning the potluck, the 61A staff decided to try and guess the number of people that would show up. In order to do this, they decided to define a new abstract data type to record everyone's predictions. Of course, the 61A staff is bad at computer science, so they need your help to make this work!

(a) (2 pt) We want to make a `prediction` abstract data type that will record both a person's name as well as their guess for the number of attendees. Based on the provided constructor `make_prediction`, fill in the definitions for the `get_name` and `get_guess` selectors.

```
def make_prediction(name, guess):
    return (name, guess)


def get_name(prediction):
    """Gets the name of the person who made the given prediction.

    >>> get_name(make_prediction('eric', 25))
    'eric'
    """

    return prediction[0]


def make_prediction(name, guess):
    return (name, guess)


def get_guess(prediction):
    """Gets the number of attendees that this prediction expected to show up
    to the potluck.

    >>> get_guess(make_prediction('eric', 25))
    25
    """

    return prediction[1]
```

(b) **(5 pt)** Now complete the `print_winner` function. It takes a sequence of `predictions` and the actual number of attendees, and prints a congratulatory message based on whose guess was closest. You may assume that the sequence of `predictions` is non-empty. Ties should go to the person whose `prediction` appears earliest in the sequence. *Remember to respect data abstraction.*

```
def print_winner(predictions, correct_num):
    """Given a sequence of predictions (predictions) and the actual number of
    attendees (correct_num), print the message '___ is the winner', where the
    blank is filled in with the name of the person who made the winning
    prediction.

    >>> albert_pred = make_prediction('albert', 10000)
    >>> brian_pred = make_prediction('brian', 85)
    >>> mark_pred = make_prediction('mark', 97)
    >>> preds = (albert_pred, brian_pred, mark_pred)
    >>> print_winner(preds, 83)
    brian is the winner
    >>> preds2 = (make_prediction('rohan', 90), make_prediction('jeffrey', 70))
    >>> print_winner(preds2, 80)
    rohan is the winner
    """

    closest = predictions[0]
    for pred in predictions[1:]:
        if abs(get_guess(pred) - correct_num) < abs(get_guess(closest) - correct_num):
            closest = pred
    print(get_name(closest), 'is the winner!')
```

6. (12 points) Learning to count

Steven likes to have a timer with him during lectures so that he knows how much time is left until the end of lecture. He has a timer he really likes that counts the number of seconds that have elapsed since the beginning of lecture.

Unfortunately, it turns out that the timer he bought was manufactured before humans had discovered the number six! The timer works normally, except it skips every number containing a 6 as one of its digits. For example, here are the first twenty numbers displayed by this timer:

```
0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21
```

In this example, note how it skips the numbers 6 and 16, because they both have at least one digit that is a 6. This means that when the timer displays 21, in reality only 19 seconds have passed!

Obviously, the way the timer is now isn't very helpful for Steven. Help him solve his problem by writing a function to compute the true number of seconds that have elapsed in his lectures.

For this entire problem, do not use any loop statements. Use recursion only.

(a) (3 pt) First, complete the `has_six` helper function, which takes an integer and returns whether or not said integer has a 6 as one of its digits. **Do not use any loop statements. Use recursion. Additionally, do not convert `n` to a string.**

```
def has_six(n):
    """Determines whether the integer n has a 6 as one of its digits.

    >>> has_six(123)
    False
    >>> has_six(567)
    True
    """

    if n == 0:
        return False
    return n % 10 == 6 or has_six(n // 10)
```

(b) (4 pt) Now, use your `has_six` function to complete the `previous` function. `previous` takes an integer `n` and determines the number that would have appeared *before* it on the timer. In other words, it determines the largest integer less than `n` that does not have a 6 as any of its digits. You may assume that `n` will always be a positive integer. **Once again, do not use any loop statements. Use recursion.**

Note: you may assume that you have a working version of `has_six`. You can receive full credit on this section without completing part (a).

```
def previous(n):
    """Determines the number that showed on the timer just before n.

    >>> previous(3)
    2
    >>> previous(7)
    5
    >>> previous(70)
    59
    """
    if not has_six(n - 1):
        return n - 1
    return previous(n - 1)
```

(c) (5 pt) Now, use your `previous` function to complete the `num_seconds` function, which takes an integer representing the number shown on the timer and returns the actual number of seconds that have elapsed. **As before, do not use any loop statements. Use recursion.**

Note: you may assume that you have a working version of `previous`. You can receive full credit on this section without completing part (b).

```
def num_seconds(n):
    """Based on the number currently displayed on the timer, n, returns the true
    number of seconds that have elapsed.

    >>> num_seconds(8) # skips 6
    7
    >>> num_seconds(20) # skips 6 and 16
    18
    """
    if n == 0:
        return 0
    return 1 + num_seconds(previous(n))
```

7. (0 points) Extra credit

In the box below, write a positive integer. The student who writes the lowest *unique* integer will receive one extra credit point. In other words, write the smallest positive integer that you think *no one else* will write.

This is the end of the test. Feel free to use the rest of the space for scratch work. You could also draw us a picture, if you're so inclined!

Comments

Problem 1

Grading: 1 pt each.

Special exceptions:

- First row: 0 points for "function, function," otherwise 1 point for any incorrect answer if the answer was written in both boxes
- Last row: 0 points for "error, error," 1 point for using `True` instead of 25

Problem 2

Part A

Grading: 2 pts total

- 1 point for correctly using a lambda function with no parameters
- 1 point for filling in a single parameter and correctly constructing a list

Common Mistakes: By far the most common mistake was forgetting to include the extra lambda function. Without it, the last set of parentheses in the call expression produces an error. Many people also tried to construct a list by using `list(x, y)`, which doesn't work. The `list` constructor expects an iterable as its single argument, so passing it the values individually does not work.

Part B

Grading: 3 pts total

- 1 point for ensuring `foxes` doesn't throw an error (usually by filling it in as the parameter)
- 1 point for calling the `love` function with argument `'zedd'`, then indexing the result at 2
- 1 point for calling the final lambda to produce the tuple

Common Mistakes: The easiest way to deal with the unbound variable `foxes` was to make it the parameter. Other approaches were possible, but most attempts didn't work out:

- Ignoring `foxes` entirely and just writing additional call expressions afterward causes a syntax error.
- Putting a comma after `foxes` wasn't allowed, since it creates a tuple. However, even if it were allowed, it *still* wouldn't work, since the comma would actually make a tuple containing a lambda and whatever expression came after the comma. This results in a "tuple isn't callable" error.

One solution that *did* work was to use a ternary operator, like so:

```
(lambda apple, banana: foxes if False else apple('zedd')[2]())(love, 'clarity')
```

We gave solutions like these full credit, as long as there were no other errors.

Problem 3

Grading: 3 pts, all or nothing

This exact behavior was actually discussed in lecture. Because the definition of `fact` makes a recursive call to `fact` within its body, the traced function only works as expected if we bind it to the name `fact`. In this case, since we bound it to `new_fact` instead, the recursive calls hit the original, untraced function.

Problem 4

Part A

Grading: 5 pts

Correctly drawing the first frame resulting from the call to `thrift(2, 3)` guaranteed you 2 points. In particular, you were required to create a `thrift` frame, add the `x` and `y` parameters, bind them to the correct values, and define and bind `inner` to a function. We refer to this as the baseline for this problem.

Each mistake (such as binding a variable to the wrong value) subtracted points. As mentioned before, if you met the baseline requirements, you could not get penalized below 2 points.

Additional rubric clarifications are available on pandagrader.

Common Mistakes: Many students used the incorrect value of `x`, usually arising due to an incorrect parent for `foo`. Some students also included an incorrect binding for `x` in the `foo` frame.

Many students also forgot to create the binding for `shop` in the global frame.

Finally, a lot of people added an extra "return value" entry in the global frame. Remember that only function calls have return values, and the global frame is not created by a function call.

Part B

Grading: 6 pts

This problem was intended as a simplification of the way the autograder for the course works!

Once again, there was a baseline of 2 points for this part. To get those points, you needed to do the following: draw a frame called `make_test`, add the parameters `num` and `checker`, bind the correct arguments to the parameters, and create the variable `test`, which points to a function with intrinsic name test.

Again, each mistake subtracted some points, and if you met the baseline requirements your point total could not fall below 2.

Additional rubric clarifications are available on pandagrader.

Common Mistakes: By far the most common mistake was an incorrect parent for the lambda function. Its parent is the global frame, since its definition occurs there (remember the argument to `test_q5` is evaluated before the call occurs). When done correctly, the value of `num` is 2, which causes the return values to be `False`.

Other common mistakes:

- Forgetting to create a frame for the lambda call
- Including a binding for `num` in the lambda frame
- Binding `test_q5` to the function with intrinsic name `make_test`
- Incorrect parent for the `test` frame
- No frame for the call to `q5_checker`

Problem 5

Part A

Grading: 2 pts

The two selectors were graded all or nothing, 1 point per selector.

Common Mistakes: Many people tried to slice (`prediction[1:]`) for `get_guess`, which returns a one-element tuple instead of the actual value itself. Many people also indexed into `make_prediction`, which is a function, instead of using the prediction itself.

Part B

Grading: 5 pts

Most solutions were graded using the following rubric:

- 1 pt for attempting to keep track of the closest guess in some form
- 1 pt for keeping track correctly
- 1 pt for attempting to loop over `predictions`
- 1 pt for hitting each element of `predictions` correctly in the loop
- 1 pt for no minor errors

Common Mistakes: One common way of losing the "keeping track correctly" point was by initializing a best name and best guess, but only updating one of the two in each iteration. Other students lost the point by comparing absolute guesses to distances.

A number of students treated `predictions` like an rlist and made a recursive function. We stated that `predictions` was a sequence, so assuming an rlist was not allowed. In these scenarios, we usually gave the point for attempting to loop, but not for looping correctly.

Many students also tried to index into predictions directly, using the fact that they're tuples. While this works with this particular representation, this is a data abstraction violation, and we often deducted points for this. A completely correct solution made use of the `get_name` and `get_guess` selectors instead.

Problem 6

We hoped that with this problem we could show you why it's so important to use helper functions! Without the use of helper functions, `num_seconds` is a pretty tricky function to write. However, by defining useful helper functions along the way, we can write each function in just 3 lines each!

Part A

Grading: 3 pts

- 1 pt for having a reasonable base case
- 1 pt for having a correct base case
- 1 pt for a correct recursive call, regardless of base case

Common Mistakes: One common mistake was not having a complete base case. In particular, many solutions were unable to handle `has_six(0)`. Solutions like these lost the point for "correct base case."

A rarer, but still relatively common mistake was to check if a number was divisible by 6. This is a completely different problem. For instance, 12 is divisible by 6, but `has_six(12)` should be `False`, since none of the digits are 6.

Part B

Grading: 4 pts

Most solutions were graded using the following rubric:

- 1 pt for using `has_six` correctly
- 1 pt for making a recursive call to `previous`
- 1 pt for passing `n - 1` as the argument to `previous`
- 1 pt for correct base case

Exceptions were handled on a case-by-case basis. Deviations that do not affect the correctness of the result (such as unnecessary base cases) were not deducted points.

Common Mistakes:

- Switching the return values of the if and else cases
- Forgetting to return the result of the recursive call
- Forgetting to make a recursive call at all
- Skipping numbers, or forgetting to advance

Part C

Grading: 5 pts

There were two major approaches to this problem. One was the recursive "count the number of ticks" solution that was provided. Another approach was to count from `n` to `0`, counting the number of sixes along the way, then subtracting that from `n` to get the final count. Both approaches were equally valid and received full credit.

This question was graded on a holistic scale. That is, rather than awarding points for individual components of the solution, we evaluated the solution as a whole. The categories were as follows:

5 points: A correct solution.

4 points: A mostly correct solution with minor errors. Some common minor errors were off-by-one errors, forgetting to call helper functions after defining them, and not including 0 in your base case.

3 points: Solutions in this category usually indicated a clear "game plan," but with major execution flaws. For the recursive tick-counting solution, this generally meant a good recursive call but minor problems using the result of the recursive call. For the six-counting solution, this generally meant that the correct structure (such as a helper function) was there, but there were major problems with the solution (such as attempting to do nonlocal assignment without using the `nonlocal` statement).

2 points: Solutions in this category indicate a basic idea of how to solve the problem, but with many of the important details missing. For the tick-counting case, this meant making recursive calls but incorrectly combining the result of successive recursive calls. (Many solutions simply returned `n` every time!) For the six-counting case, this usually took the form of counting sixes using a local variable, instead of setting up a helper function like was necessary.

1 point: Generally speaking, solutions in this category had a valid base case, or were consistent with their range (always returning a number), but did not accomplish much else.

0 points: Did not meet any of the requirements of the above categories, or did not follow the instructions (no loop statements).