
CS 61A Structure and Interpretation of Computer Programs

Spring 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	
<i>All the work on this exam is my own. (please sign)</i>	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Total
/12	/12	/6	/12	/8	/50

1. (12 points) Call Me Maybe

For each of the following call 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 add, mul
def mulled(x, y):
    return mul(x, add(y, x))

def fauxpose(f, g):
    print('maybe')
    def h(x, y):
        f(x, y)
        return g(x, y)
    return h
```

Expression	Evaluates to	Interactive Output
mulled(5, 5)	50	50
1/0	ERROR	ERROR
mulled(4, 1)	20	20
add(mulled(3, 2), print(4))	Error	4 Error
print(3, print(5, print(1)))	None	1 5 None 3 None
fauxpose(add, mul)(3, 2)	6	maybe 6
fauxpose(mul, print)(4, 1)	None	maybe 4 1
fauxpose(fauxpose, mulled)(2, 5)	14	maybe maybe 14

2. (12 points) We Are All Environmentalists

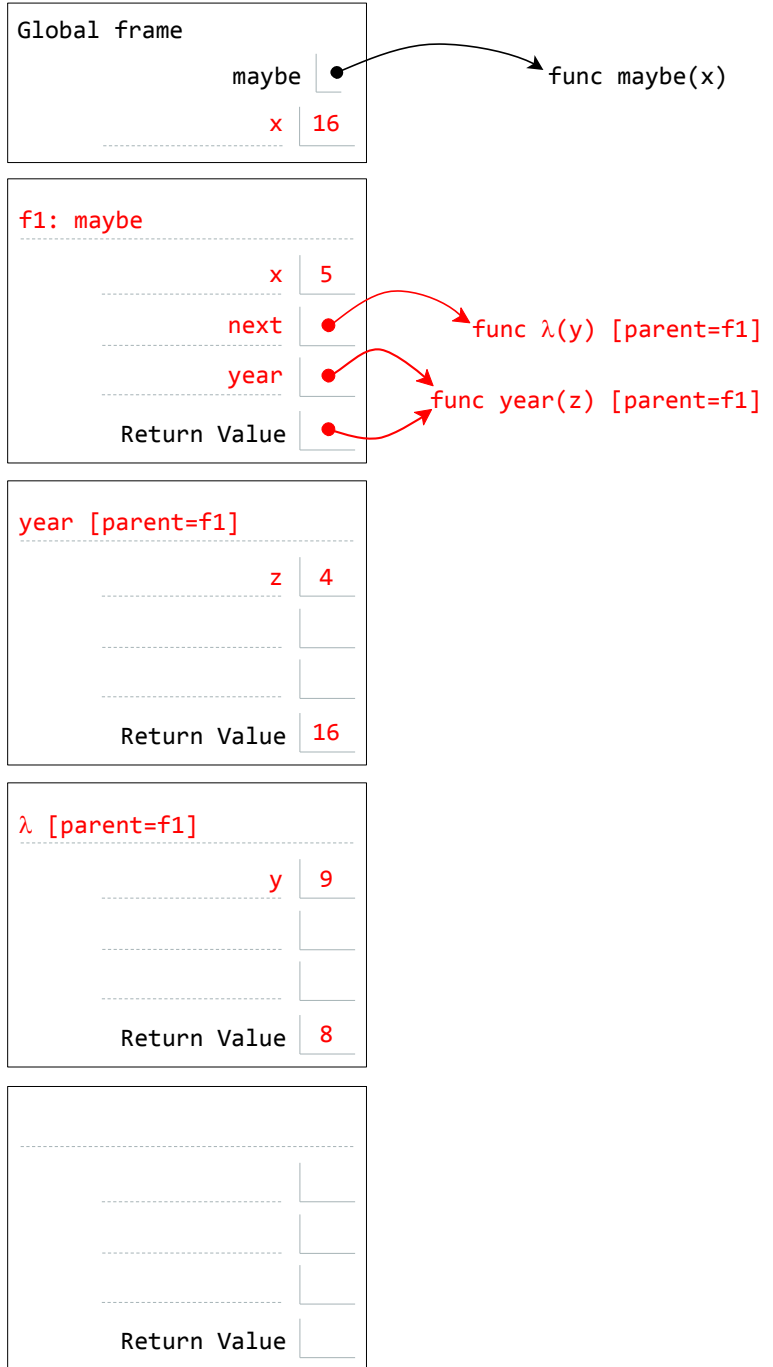
(a) (6 pt) Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. You need only show the final state of each frame. You may not need to use all of the spaces or frames.

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 maybe(x):
    next = lambda y: y - 1
    x += 3
    def year(z):
        return next(z + x) * 2
    return year

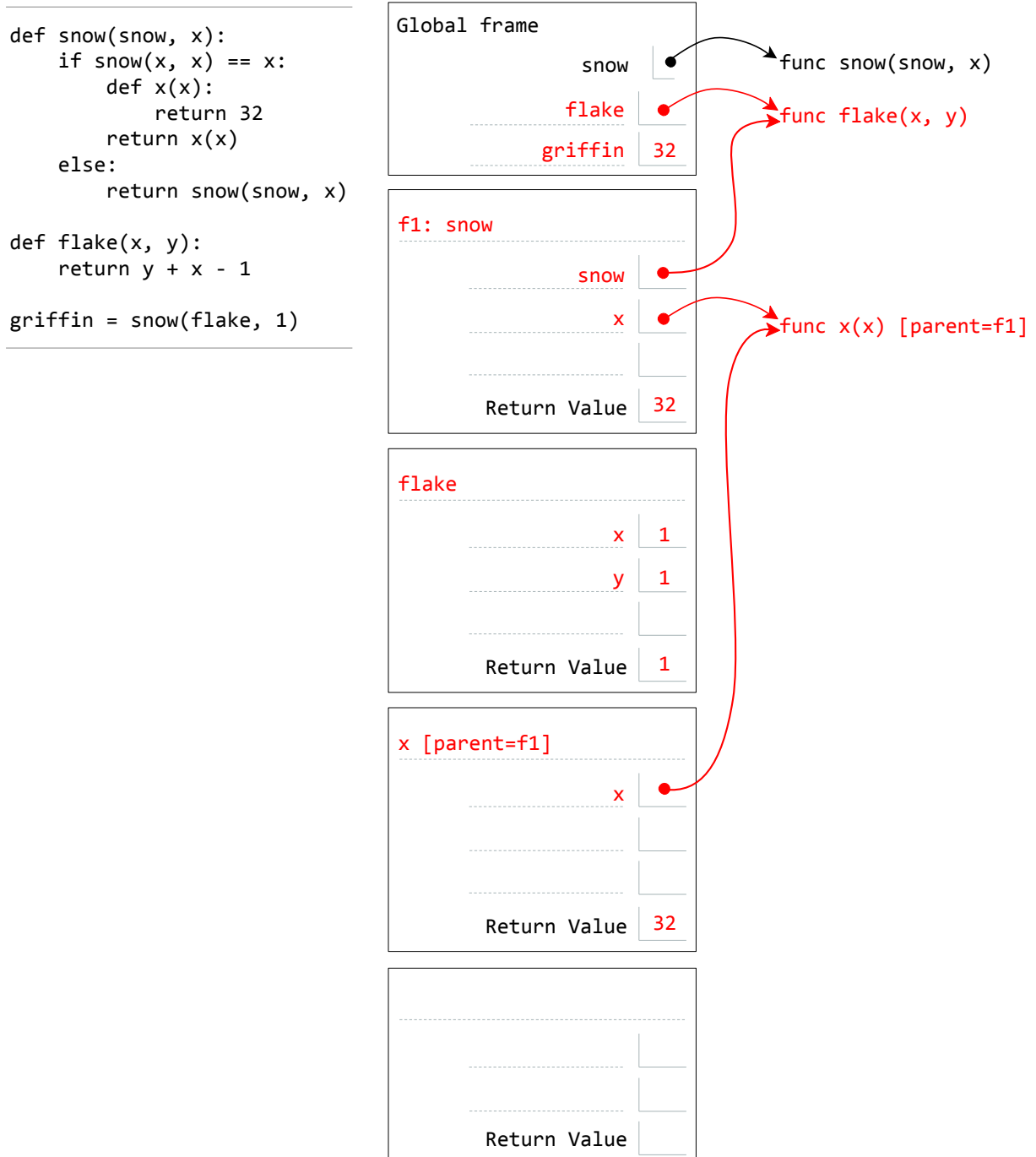
x = maybe(2)(4)
```



(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.



3. (6 points) A Higher Order of Protection

Louis Reasoner is making a web application, and he wants to secure it. (Good for him!) One of the ways he wants to secure it is through checking to make sure that the user is an admin when it tries to visit certain confidential pages. So, being a silly programmer, he does the following.

```
def delete_everything(is_admin, request):
    if not is_admin:
        print('ERROR: not admin')
        return
    confirmation = do_bad_stuff(request) # BAD STUFF HAPPENS HERE
    return confirmation

def steal_credit_card_info(is_admin, request):
    if not is_admin:
        print('ERROR: not admin')
        return
    cc_info = hack_a_shaq(request) # DO SOME 1337 HAXORING
    return cc_info
```

However, Alyssa P. Hacker comes across this code, and realizes that there is a better way to do this using higher-order functions! She modifies the above as follows.

```
def delete_everything(request):
    confirmation = do_bad_stuff(request) # BAD STUFF HAPPENS HERE
    return confirmation
delete_everything = protect_me(delete_everything)

def steal_credit_card_info(request):
    cc_info = hack_a_shaq(request) # DO SOME 1337 H4XORING
    return cc_info
steal_credit_card_info = protect_me(steal_credit_card_info)
```

Help her to complete the code by filling in the function below. The new code should provide the same functionality as the original code. For example, calling `delete_everything(True, my_request)` should have the same effect in both versions of the code.

You may leave lines blank if you do not need them.

```
def protect_me(fn):
    def wrapper(is_admin, request):
        if not is_admin:
            print('ERROR: not admin')
            return
        return fn(request)
    return wrapper
```

4. (12 points) Da Visors Provide Protection from Puns

An integer d is a *divisor* of another integer n if it evenly divides n , i.e. the remainder is 0 when dividing n by d . The divisors of n include 1 and n itself.

- (a) Complete the function below to compute the number of positive divisors of a positive integer. Fill in the blanks. You may leave a line blank if you do not need it.

```
def num_divisors(n):
    """Computes the number of positive divisors of a positive integer.

    >>> num_divisors(4)    # 1, 2, and 4
    3
    """
    i, count = 1, 0
    while i <= n:
        if n % i == 0:
            count = count + 1
        i = i + 1
    return count
```

- (b) Write a function that computes the sum of the positive divisors of a positive integer.

```
def sum_divisors(n):
    """Computes the sum of the positive divisors of a positive integer.

    >>> num_divisors(4)    # 1 + 2 + 4
    7
    """
    i = 1
    sum = 0
    while i <= n:
        if n % i == 0:
            sum += i
        i = i + 1
    return sum
```

- (c) A positive integer n is called *abundant* if the sum of its divisors (except n itself) is strictly larger than n . It is called *perfect* if the sum of its divisors (except n itself) is exactly equal to n . Finally, n is deficient if the sum of its divisors (excluding n) is strictly less than n . Write a function that returns the string 'abundant' if the input n is abundant, 'perfect' if n is perfect, and 'deficient' if n is deficient. You may call `sum_divisors` and assume that it works correctly.

```
def describe(n):
    """Returns whether n is abundant, perfect, or deficient.

    >>> describe(4)      # 1 + 2 < 4
    'deficient'
    """
    divisors = sum_divisors(n) - n
    if divisors == n:
        return 'perfect'
    elif divisors > n:
        return 'abundant'
    else:
        return 'deficient'
```

5. (8 points) Lambda the Free

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

```

square = lambda x: x * x

def muckluck(x):
    square(square(x))

def apply(f, x):
    return f(x)

def apply_many(f, x, n):
    while n > 0:
        x = f(x)
    return x

def wut(f):
    return lambda f: f(x)

def pair(x, y):
    return lambda k: x if k == 0 else y

```

For each of the following call expressions, write the value to which it evaluates. If the value is a function value, write FUNCTION. If evaluation causes an error, write ERROR. If evaluation would run forever, write FOREVER.

Expression	Evaluates to
square	Function
muckluck(2)	None
apply(pair, 3)	Error
wut(square)	Function
pair(3, 4)(1)	4
pair(apply, square)(0)	Function
apply_many(square, 3, 0)	3
apply_many(square, 3, 2)	Forever

Comments

Problem 1

Grading: 1 pt each.

Common Mistakes: Wrong order for output. Not printing `None`. Evaluating to “nothing” instead of `None`. (`None` is an actual Python value.) Missing the second “maybe” that results from `fauxpose(fauxpose, mulled)(2, 5)`. (A “maybe” is printed each time `fauxpose` is applied.) Incorrect behavior for the `print` function.

Suggested Approach: This questions requires a full understanding of the execution/evaluation procedure for each statement/expression. For example, a definition statement is executed by creating a new function value and binding the name to that value in the current frame. The most important rule is that for call expressions, which is as follows:

- Evaluate the operator and operands from left to right.
- Apply the function that the operator evaluates to on the argument values that result from evaluating the operands.
 - Create a new frame.
 - Bind the parameter names to the argument values in that frame.
 - Execute the body in the context of the new environment.
- Evaluate to the return value of the applied function.

These rules, along with the rules for the other statements and expressions, are enough to answer this problem. You should not guess at the answer or take any shortcuts; just apply the rules mechanically.

Problem 2

Part A

Grading: 6 pts

Common Mistakes: Leaving out parent frames, or recording the wrong parent. Using the name `next` instead of `λ` for the lambda function. Wrong return value for `maybe`.

Part B

Grading: 6 pts

Common Mistakes: Calling the `snow` function recursively instead of looking the name `snow` up in the current environment. Neglecting to draw frames for every function application. Improper rebinding of an existing name (e.g. not updating the binding at all, are adding a second binding for the name in the same frame; a frame can only contain a single binding for each name). Not binding a parameter name upon function application. Binding the the name `griffin` to the `snow` function instead of its return value.

Suggested Approach: As in problem 1, understanding execution and evaluation procedures is crucial. Again, don’t take any shortcuts: follow the procedures mechanically and exactly. Make sure to look up each name in the current environment each time the name is encountered.

Problem 3

Grading: 6 pts

Common Mistakes: Leaving out one of the two arguments of the locally-defined function. Not returning that function. Not including the `is_admin` check in the locally-defined function. Not calling the original function `fn`. Defining the local function with the name `fn`.

Suggested Approach: There is a lot of reading in this problem, so it is important to go through it carefully and identify what is required. In reading through the problem, you should start by finding all uses of the `protect_me` function to learn as much as you can about it. In particular, we see that the return value of `protect_me` is assigned to the name `delete_everything`, which is subsequently called. This implies that `protect_me` must return a function, which likely means it should be defining a function locally. Furthermore, the function bound to `delete_everything` is called on two arguments, so that local function must have two parameters. Looking at Louis Reasoner's code, we see that the `is_admin` check happens every time `delete_everything` is called, so we need to make sure that check happens in the function that gets assigned to `delete_everything` in Alyssa's code. Finally, that function must also call `do_bad_stuff` and return `confirmation`, so we need to make sure that what gets assigned to `delete_everything` does that as well. Now we have all the information we need to write `protect_me`, so only then should we proceed to do so.

As with the previous problems, a systematic approach is necessary, and you should not take any shortcuts. Do not start writing code until you understand the problem completely.

Problem 4

Grading: 3 pts for (a), 5 pts for (b), 4 pts for (c)

Common Mistakes: Printing instead of returning. Improper indentation. Using `=` instead of `==`.

Suggested Approach: Make sure to read each doctest carefully to understand what each function should compute and what it should return. Once you write a function, test it on representative examples by actually going through the execution procedure and making sure the right result is computed.

Problem 5

Grading: 1 pt each

Common Mistakes: Missing the lack of a return statement in `muckluck`. Evaluating the body of a lambda function when it is defined.

Suggested Approach: Same as problems 1 and 2.