CS 61A Fall 2015

Structure and Interpretation of Computer Programs

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" $\times$ 11" crib sheet of your own creation and the official CS 61A midterm 1 study guide.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

Last name	
First name	
Student ID number	
BearFacts email (`_@berkeley.edu`)	
TA	
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)**	

1. (12 points) Evaluators Gonna Evaluate

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "Error".

Hint: No answer requires more than 5 lines. (It's possible that all of them require even fewer.)

The first two rows have been provided as examples.

Recall: The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

Assume that you have started `python3` and executed the following statements:

```
def jazz(hands):
    if hands < out:
        return hands * 5
    else:
        return jazz(hands // 2) + 1

def twist(shout, it, out=7):
    while shout:
        shout, out = it(shout), print(shout, out)
    return lambda out: print(shout, out)

hands, out = 2, 3
```

Expression	Interactive Output
`pow(2, 3)`	8
`print(4, 5) + 1`	4 5 Error
`print(None, print(None))`	None None None
`jazz(5)`	11
`(lambda out: jazz(8))(9)`	12
`twist(2, lambda x: x-2)(4)`	2 7 0 4
`twist(5, print)(out)`	5 5 7 None 3
`twist(6, lambda hands: hands-out, 2)(-1)`	6 2 3 None 0 -1

2. (12 points) Environmental Policy

(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 may not need to use all of the spaces or frames.*

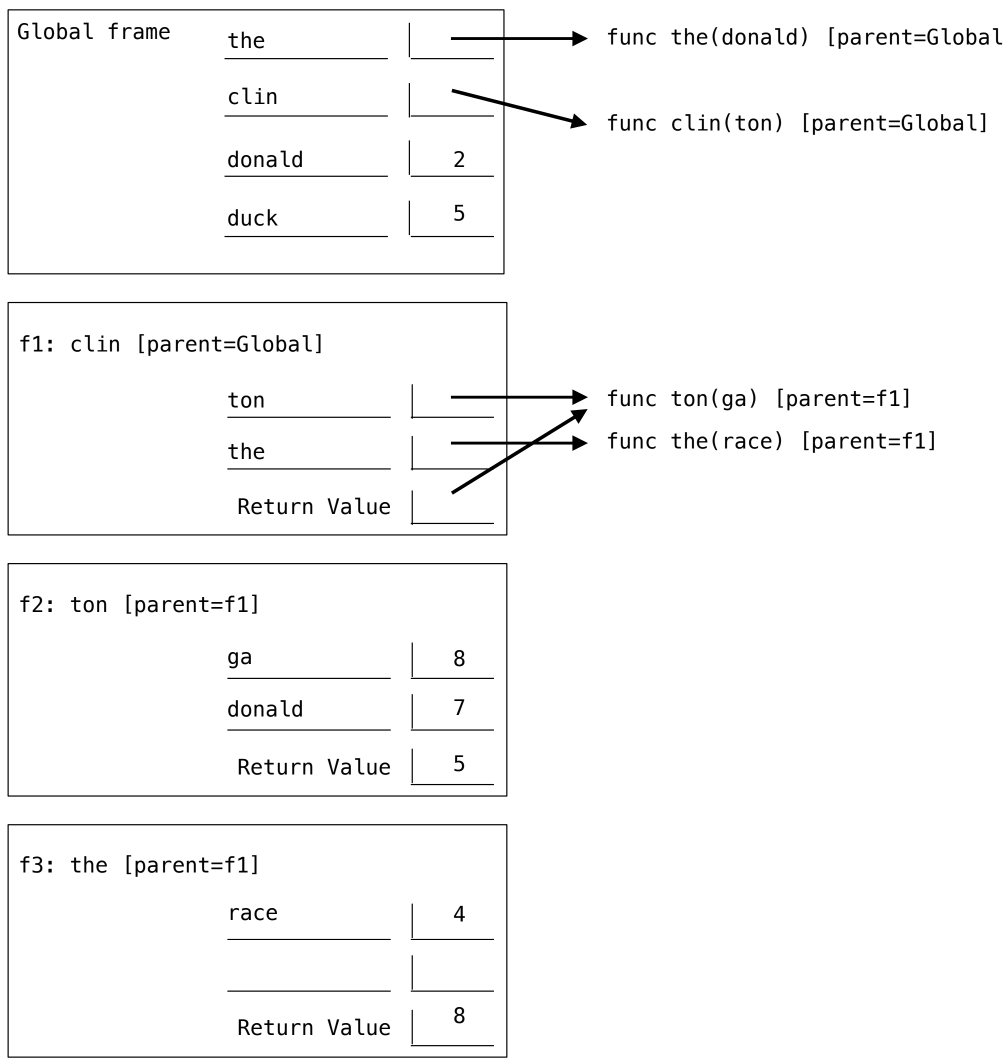
A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
def the(donald):
    return donald + 5

def clin(ton):
    def the(race):
        return donald + 6
    def ton(ga):
        donald = ga - 1
        return the(4) - 3
    return ton

donald, duck = 2, clin(the)
duck = duck(8)
```

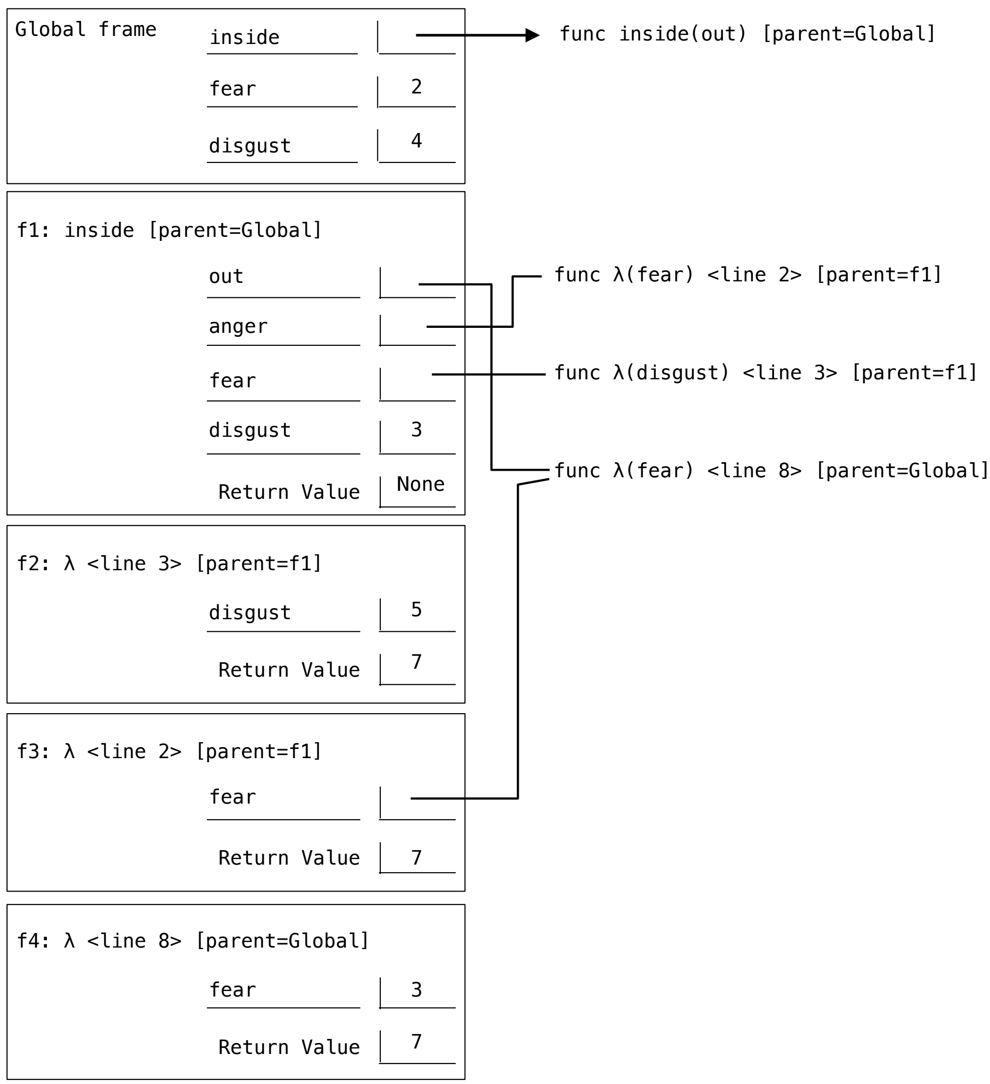


(b) **(6 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* The `<line ...>` annotation in a lambda value gives the line in the Python source of a lambda expression.

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Add all missing parents of function values.
- Show the return value for each local frame.

```
1 def inside(out):
2     anger = lambda fear: fear(disgust)
3     fear = lambda disgust: anger(out)
4     disgust = 3
5     fear(5)
6
7 fear, disgust = 2, 4
8 inside(lambda fear: fear + disgust)
```



3. (10 points) Digit Fidget

****IMPORTANT DEFINITION**** Each digit in a non-negative integer `n` has a *digit position.* Digit positions begin at 0 and count from the right-most digit of `n`. For example, in 568789, the digit 9 is at position 0 and digit 7 is at position 2. The digit 8 appears at both positions 1 and 3.

(a) (3 pt) Implement the `find_digit` function, which takes a non-negative integer `n` and a digit `d` **greater than 0 and less than 10**. It returns the *largest* (left-most) position in `n` at which digit `d` appears. If `d` does not appear in `n`, then `find_digit` returns `False`. *You may not use recursive calls.*

```
def find_digit(n, d):
    """Return the largest digit position in n for which d is the digit.

    >>> find_digit(567, 7)
    0
    >>> find_digit(567, 5)
    2
    >>> find_digit(567, 9)
    False
    >>> find_digit(568789, 8)
    3
    """

    i, k = 0, False

    while n:

        n, last = n // 10, n % 10

        if last == d:

            k = i

        i = i + 1

    return k
```

(b) (2 pt) Circle **all** values of y for which the final expression below evaluates to `True`. Assume that `find_digit` is implemented correctly. The `compose1` function appears on the left column of page 2 of your study guide.

1 2 3 4 5 6 7 8 9

```
f = lambda x: find_digit(234567, x)
compose1(f, f)(y) == y
```

(c) **(3 pt)** Implement `luhn_sum`. The *Luhn sum* of a non-negative integer n adds the sum of each digit in an *even* position to the sum of doubling each digit in an *odd* position. If doubling an odd digit results in a two-digit number, those two digits are summed to form a single digit. *You may not use recursive calls or call `find_digit` in your solution.*

```
def luhn_sum(n):
    """Return the Luhn sum of n.

    >>> luhn_sum(135)     # 1 + 6 + 5
    12
    >>> luhn_sum(185)     # 1 + (1+6) + 5
    13
    >>> luhn_sum(138743)  # From lecture: 2 + 3 + (1+6) + 7 + 8 + 3
    30
    """
    def luhn_digit(digit):

        x = digit * multiplier

        return (x // 10) + x % 10

    total, multiplier = 0, 1

    while n:

        n, last = n // 10, n % 10

        total = total + luhn_digit(last)

        multiplier = 3 - multiplier

    return total
```

(d) **(2 pt)** A non-negative integer has a *valid* Luhn sum if its Luhn sum is a multiple of 10. Implement `check_digit`, which appends one additional digit to the end of its argument so that the result has a valid Luhn sum. Assume that `luhn_sum` is implemented correctly.

```
def check_digit(n):
    """Add a digit to the end of n so that the result has a valid Luhn sum.

    >>> check_digit(153) # 2 + 5 + 6 + 7 = 20
    1537
    >>> check_digit(13874)
    138743
    """

    return 10 * n + -luhn_sum(10 * n) % 10
```

4. (6 points) Zombies!

****IMPORTANT**** In this question, assume that all of `f`, `g`, and `h` are functions that take **one** non-negative integer argument and return a non-negative integer. You *do not* need to consider negative or fractional numbers.

(a) (4 pt) Implement the higher-order function `decompose1`, which takes two functions `f` and `h` as arguments. It returns a function `g` that relates `f` to `h` in the following way: For any non-negative integer `x`, `h(x)` equals `f(g(x))`. Assume that `decompose1` will be called only on arguments for which such a function `g` exists. Furthermore, assume that there is no recursion depth limit in Python.

```
def decompose1(f, h):
    """Return g such that h(x) equals f(g(x)) for any non-negative integer x.

    >>> add_one = lambda x: x + 1
    >>> square_then_add_one = lambda x: x * x + 1
    >>> g = decompose1(add_one, square_then_add_one)
    >>> g(5)
    25
    >>> g(10)
    100
    """

    def g(x):

        def r(y):

            if h(x) == f(y):

                return y

            else:

                return r(y+1)

        return r(0)

    return g
```

(b) (2 pt) Write a number in the blank so that the final expression below evaluates to 2015. Assume `decompose1` is implemented correctly. The `make_adder` and `compose1` functions appear on the left column of page 2 of your study guide.

```
e, square = make_adder(1), lambda x: x*x

decompose1(e, compose1(square, e))(3) + 2000
```