CS 61A Fall 2016

Structure and Interpretation of Computer Programs

MIDTERM 1

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 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	
CalCentral 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. (10 points) Exeggcute

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", but include all output displayed before the error. If a function value is displayed, write "Function".

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:

```
equals = lambda a, b: a == b
nemo = lambda n: lambda: print(n)
ray = nemo(1)

def if_function(f, g, h):
    if h:
        f()
    elif h:
        f(f())
    else:
        print(5 or 6)
    g()

def dory():
    print('fish')
    return lambda: 1/0
```

Expression	**Interactive Output**
`pow(2, 3)`	8
`print(4, 5) + 1`	4 5 Error
`equals(3==4, equals(5, equals(5, 5)))`	
`print(print(print(2)), print(3))`	
`print(nemo(print(5))())`	
`if_function(nemo(3), dory, 2)`	
`if_function(dory, nemo(2), ray())`	

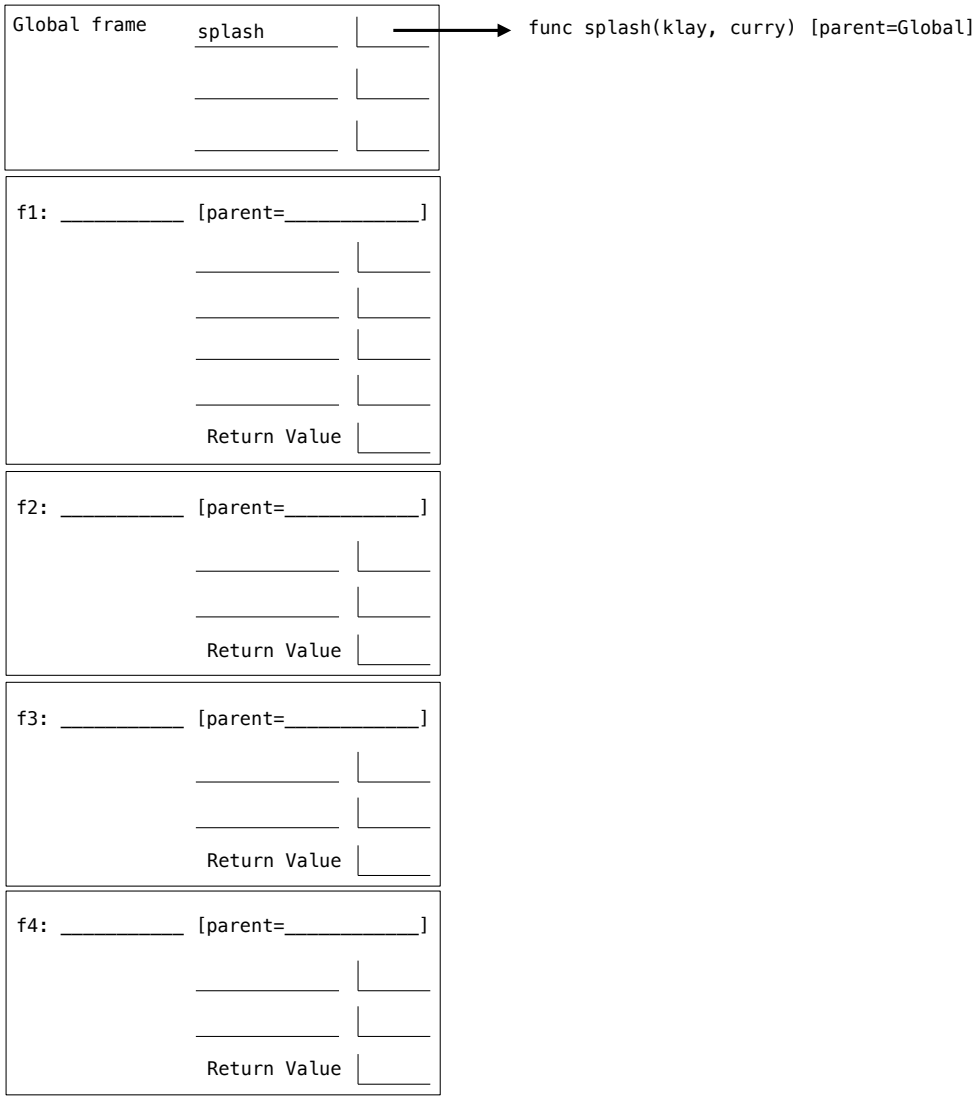
2. (8 points) Goldeen State

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.

```
1  def splash(klay, curry):
2    while curry == 3:
3      steph = klay
4      klay = lambda klay: steph(curry + 1)
5      curry = curry + 2
6      return klay
7    return curry // 5
8
9  steph = lambda klay: splash(11, curry)-3
10 steph, curry = splash(steph, 3), 30
11 steph(4)
```



3. (9 points) Countizard

(a) (6 pt) Implement `counter`, which takes a non-negative single-digit integer `d`. It returns a function `count` that takes a non-negative integer `n` and returns the number of times that `d` appears as a digit in `n`. You may not use recursive calls or any features of Python not yet covered in the course.

```
def counter(d):
    """Return a function of N that returns the number of times D appears in N.

    >>> counter(8)(8018)
    2
    >>> counter(0)(2016)
    1
    >>> counter(0)(0)
    0
    """
    def count(_____________________________________________________________):

        k = 0

        while _____________________________________________________________:


            _____________, last = _____________________________________, n % 10


            if _____________________________________________________________:


                ____________________________________________________________


        ____________________________________________________________________


    ____________________________________________________________________________
```

(b) (3 pt) Implement `significant`, which takes positive integers `n` and `k`. It returns the `k` most significant digits of `n` as an integer. These are the first `k` digits of `n`, starting from the left. If `n` has fewer than `k` digits, it returns `n`. You may not use `round`, `int`, `str`, or any functions from the `math` module. You may use `pow`, which raises its first argument to the power of its second: `pow(9, 2)` is 81 and `pow(9, 0.5)` is 3.0.

```
def significant(n, k):
    """Return the K most significant digits of N.

    >>> significant(12345, 3)
    123
    >>> significant(12345, 7)
    12345
    """
    if ____________________________________________________________________:

        return n

    return significant(_____________________________, _____________________________)
```

4. (6 points) Caterepeat

(a) (4 pt) Implement `repeat_sum`, which takes a one-argument function `f`, a value `x`, and a non-negative integer `n`. It returns the sum of $n + 1$ terms. Each term, indexed by `k` starting at 0, is the result of applying `f` to `x` repeatedly `k` times. You may assign to only one name in each of the three assignment statements. You may not use recursive calls or any features of Python not yet covered in the course.

```
def repeat_sum(f, x, n):
    """Compute the following summation of N+1 terms, where the last term
    calls F N times: x + f(x) + f(f(x)) + f(f(f(x))) + ... + f(f(...f(x)))

    >>> repeat_sum(lambda x: x*x, 3, 0) # 3
    3
    >>> repeat_sum(lambda x: x*x, 3, 1) # 3 + 9
    12
    >>> repeat_sum(lambda x: x+2, 3, 4) # 3 + 5 + 7 + 9 + 11
    35
    """
    total, k = 0, 0

    while ____________________________________________________________________:

        _______________ = ____________________________________________________

        _______________ = ____________________________________________________

        _______________ = ____________________________________________________

    return total
```

(b) (2 pt) Implement `sum_squares`, which takes a non-negative integer `n` and uses `repeat_sum` to return the sum of the squares of the first `n` positive integers. Assume `repeat_sum` is implemented correctly. You may use `pow`, which raises its first argument to the power of its second: `pow(9, 2)` is 81 and `pow(9, 0.5)` is 3.0.

```
def sum_squares(n):
    """Return the sum of the first N perfect squares.

    >>> sum_squares(0)
    0
    >>> sum_squares(3) # 1**2 + 2**2 + 3**2
    14
    >>> sum_squares(5) # 1**2 + 2**2 + 3**2 + 4**2 + 5**2
    55
    """

    f = ____________________________________________________________________

    return repeat_sum(f, 0, n)
```

5. (7 points) Multikarp

Terminology. An *order 1 numeric function* is a function that takes a number and returns a number. An order 2 numeric function is a function that takes a number and returns an order 1 numeric function. Likewise, an order n numeric function is a function that takes a number and returns an order $n-1$ numeric function.

The *argument sequence* of a nested call expression is the sequence of all arguments in all subexpressions, in the order they appear. For example, the expression `f(3)(4)(5)(6)(7)` has the argument sequence 3, 4, 5, 6, 7.

(a) (4 pt) Implement `multiadder`, which takes a positive integer `n` and returns an order n numeric function that sums an argument sequence of length n.

```
def multiadder(n):
    """Return a function that takes N arguments, one at a time, and adds them.

    >>> f = multiadder(3)
    >>> f(5)(6)(7)                # 5 + 6 + 7
    18
    >>> multiadder(1)(5)
    5
    >>> multiadder(2)(5)(6)       # 5 + 6
    11
    >>> multiadder(4)(5)(6)(7)(8) # 5 + 6 + 7 + 8
    26
    """
    assert n > 0


    if ____________________________________________________________________:


        return ____________________________________________________________


    else:


        return ____________________________________________________________
```

(b) (3 pt) Complete the expression below by writing one integer in each blank so that the whole expression evaluates to 2016. The `compose1` function appears on your midterm 1 study guide in the middle of the left column of page 2. Assume `multiadder` is implemented correctly.

```
compose1(multiadder(_______)(1000), multiadder(_______)(10)(_______))(1)(2)(3)
```