CS 61A Structure and Interpretation of Computer Programs

Fall 2019

Midterm 1

INSTRUCTIONS

- You have 1 hour and 50 minutes 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)**	

POLICIES & CLARIFICATIONS

- If you need to use the restroom, bring your phone and exam to the front of the room.
- You may use built-in Python functions that do not require import, such as `min`, `max`, `pow`, `len`, and `abs`.
- You **may not** use example functions defined on your study guide unless a problem clearly states you can.
- For fill-in-the-blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.

1. (12 points) What Would Python Display

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 evaluation would run forever, write "Forever". To display a function value, write "Function". The first two rows have been provided as examples.

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

Assume that you have first started `python3` and executed the statements on the left.

```
def mint(y):
    return print(-2)

def snooze(e, f):
    if e and f():
        print(e)
    if e or f():
        print(f)
    if not e:
        print('naughty')

def lose():
    return -1

def alarm():
    print('Midterm')
    1 / 0
    print('Time')

def sim(b, a):
    while a > 1:
        def sc(ar):
            a = b + 4
            return b
        a, b = a // 2, b - a
        print(a)
    print(sc(b - 1), a)

pumbaa = lambda f: lambda x: f(f(x))
pumbaa = pumbaa(pumbaa)
rafiki = 1
timon = lambda y: y + rafiki
rafiki = -1
```

	Expression	Interactive Output
	`pow(10, 2)`	100
	`print(4, 5) + 1`	4 5 Error
(2 pt)	`print(mint(print))`	
(3 pt)	`print(snooze(1, lose))`	
(3 pt)	`snooze(print(1), alarm)`	
(2 pt)	`sim(3, 3)`	
(2 pt)	`pumbaa(timon)(5)`	

2. (6 points) Environmental Studies

Fill in the environment diagram that results from executing the code on the right 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 oski(oski):
    x = 1
    if oski(2) > x:
        return oski
bear = lambda z: (lambda y: z+3)(x+4)
x, z = 11, 12
x = oski(bear)
```

```
Global frame    oski ________ | ______  ---->  func oski(oski) [parent=Global]
                _____________ | ______
                _____________ | ______
                _____________ | ______

f1: __________ [parent=__________]
                _____________ | ______
                _____________ | ______
                _____________ | ______
                 Return Value | ______

f2: __________ [parent=__________]
                _____________ | ______
                _____________ | ______
                 Return Value | ______

f3: __________ [parent=__________]
                _____________ | ______
                _____________ | ______
                 Return Value | ______
```

3. (5 points) You Again

Implement `again`, which takes a function `f` as an argument. The `again` function returns the smallest non-negative integer `n` for which `f(n)` is equal to `f(m)` for some non-negative `m` that is less than `n`. Assume that `f` takes non-negative integers and returns the same value for at least two different non-negative arguments.

Constraints:

- Lines numbered 2, 4, and 5 <u>must</u> begin with either `while` or `if`.
- Lines numbered 6 and 7 <u>must</u> contain either `return` or `=`.

```
def parabola(x):
    """A parabola function (for testing the again function)."""
    return (x-3) * (x-6)

def vee(x):
    """A V-shaped function (for testing the again function)."""
    return abs(x-2)

def again(f):
    """Return the smallest non-negative integer n such that f(n) == f(m) for some m < n.

    >>> again(parabola)  # parabola(4) == parabola(5)
    5
    >>> again(vee)       # vee(1) == vee(3)
    3
    """
```

```
1       n = 1
2       ----------------------------------------------------------------------------:
3           m = 0
4           ------------------------------------------------------------------------:
5               --------------------------------------------------------------------:
6                   ----------------------------------------------------------------
7               --------------------------------------------------------------------
8           n = n + 1
```

4. (17 points) Ups and Downs

Definition. Two adjacent digits in a non-negative integer are an *increase* if the left digit is smaller than the right digit, and a *decrease* if the left digit is larger than the right digit.

For example, 61127 has 2 increases ($1 \to 2$ and $2 \to 7$) and 1 decrease ($6 \to 1$).

You may use the `sign` function defined below in all parts of this question.

```
def sign(x):
    if x > 0:
        return 1
    elif x < 0:
        return -1
    else:
        return 0
```

(a) (5 pt) Implement `ramp`, which takes a non-negative integer `n` and returns whether it has more *increases* than *decreases* when reading its digits from left to right (see the definition above).

```
def ramp(n):
    """Return whether non-negative integer N has more increases than decreases.

    >>> ramp(123)   # 2 increases (1 -> 2, 2 -> 3) and 0 decreases
    True
    >>> ramp(1315)  # 2 increases (1 -> 3, 1 -> 5) and 1 decrease (3 -> 1)
    True
    >>> ramp(176)   # 1 increase (1 -> 7) and 1 decrease (7 -> 6)
    False
    >>> ramp(5)     # 0 increases and 0 decreases
    False
    """

    n, last, tally = ______________________________ , _______________________________ , 0

    while n:

        n, last, tally = n // 10, n % 10, _______________________________________________


    return _____________________________________________________________________________
```

(b) (3 pt) Implement `over_under`, which takes a number `y` and returns a function that takes a number `x`. This function returns 1 if `x` is greater than `y`, 0 if `x` equals `y`, and -1 if `x` is less than `y`.

You may <u>not</u> use `if`, `and`, or `or`.

```
def over_under(y):
    """Return a function that takes X and returns:
         -1 if X is less than Y
          0 if X is equal to Y
          1 if X is greater than Y

    >>> over_under(5)(3)  # 3 < 5
    -1
    >>> over_under(5)(5)  # 5 == 5
    0
    >>> over_under(5)(7)  # 7 > 5
    1
    """
    return _____________________________________________________________________________
```

Read this first. The `process` function below uses `tally` and `result` functions to analyze all adjacent pairs of digits in a non-negative integer `n`. A `tally` function is called on each pair of adjacent digits.

```
def process(n, tally, result):
    """Process all pairs of adjacent digits in N using functions TALLY and RESULT."""
    while n >= 10:
        tally, result = tally(n % 100 // 10, n % 10)
        n = n // 10
    return result()
```

(c) **(6 pt)** Implement `ups`, which returns two functions that can be passed as `tally` and `result` arguments to `process`, so that `process` computes whether a non-negative integer `n` has exactly `k` *increases*.

Hint: You can use `sign` from the previous page and the built-in `max` and `min` functions.

```
def ups(k):
    """Return tally and result functions that compute whether N has exactly K increases.

    >>> f, g = ups(3)
    >>> process(1200849, f, g)    # Exactly 3 increases: 1 -> 2, 0 -> 8, 4 -> 9
    True
    >>> process(94004, f, g)      # 1 increase: 0 -> 4
    False
    >>> process(122333445, f, g)  # 4 increases: 1 -> 2, 2 -> 3, 3 -> 4, 4 -> 5
    False
    >>> process(0, f, g)          # 0 increases
    False
    """
    def f(left, right):


        return ____________(____________________________________________________________)


    return ____________________________________ , ____________________________________
```

(d) **(3 pt)** Implement `at_most`, which returns `True` if the number of *increases* in a non-negative integer `n` is less than or equal to `k`, and `False` otherwise. Assume `ups` is implemented correctly. You may use any of the functions from previous parts of this question: `sign`, `ramp`, `over_under`, and `process`.

```
def at_most(n, k):
    """Return whether non-negative integer N has at most K increases.

    >>> at_most(567, 3)
    True
    >>> at_most(567, 2)
    True
    >>> at_most(567, 1)
    False
    """
    result = ____________________________________________________________________________

    while k >= 0:

        a, b = ups(k)

        k, result = k - 1, ______________________________________________________________

    return result
```