CS 61A Structure and Interpretation of Computer Programs
Spring 2019

MIDTERM 1

INSTRUCTIONS

- You have 55 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 guides 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. (10 points, 15 minutes) What Would Python Display *(All are in Scope: WWPD, Iterators, Generators, Lambda Expressions, Higher-Order Functions)*

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.

```
n = 1
f = lambda: n
g = f
n = 2
```

```
x, y, loops = 0, 21, 0
while y:
    loops += 1
    if x <= y:
        if x % 2 == 1:
            x, y = y, x
        elif x == 0:
            x += (2 ** 2)
        else:
            x = x + 3
            y = y // 2
    else:
        y = 0
```

```
def foo(bar, z):
    def bar(w):
        z = w + 2
        return z
    print(bar(z))
    print(z)
    return bar
```

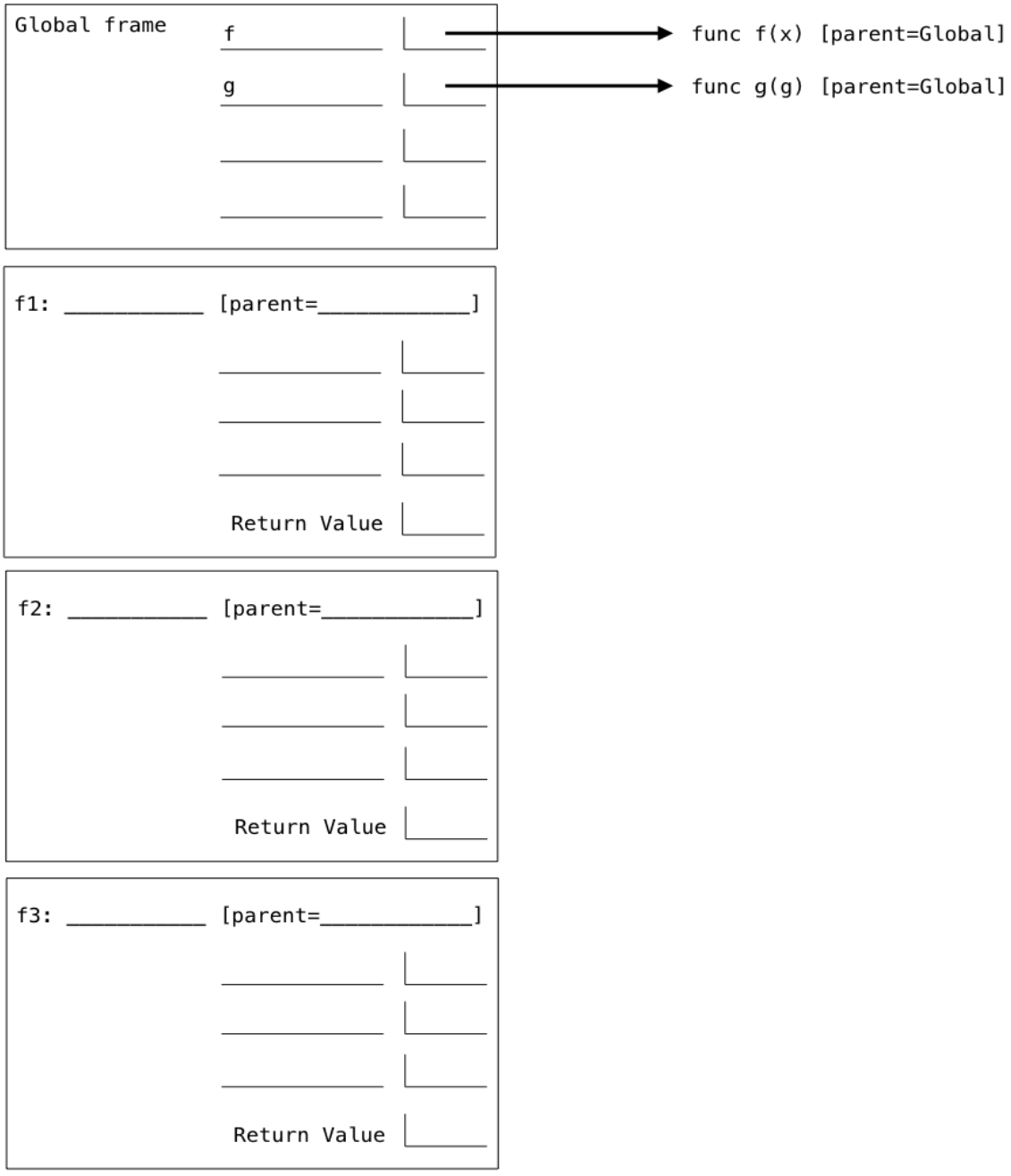
Expression	**Interactive Output**
`pow(10, 2)`	100
`print(4, 5) + 1`	4 5 Error
`print(1, print(print(2), 3 or 4 // 0))`	
`g()`	
`x`	
`loops`	
`foo(lambda m: m + 1, 4)(7)`	

2. (10 points, 15 minutes) Environment Diagram

(All are in Scope: Environment Diagrams, Higher-Order Functions)
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.

```
1  def f(x):
2    f = 10
3    z = 100
4    x = g(x)(f)
5    def f(x):
6      return x + y
7    return f
8
9  def g(g):
10   return lambda g: g + z
11
12 z = 3
13 f = f(5)
```



3. (10 points, 25 minutes) Zip It *(At least one of these is out of Scope: Control, Self Reference, Higher-Order Functions)*

We would like to create a function `make_zipper` that takes two functions `f1(x)` and `f2(x)` and a "zipper sequence", which is a number that contains a series of 1s and 2s. It returns a function that is the equivalent of `f1(f2(f2(...f1(x)...)))` in which the exact sequence of `f1`s and `f2`s is given by the digits of the sequence. As an example, if the sequence were `1211`, that would mean return a function of `x` that is the equivalent to `f1(f2(f1(f1(x))))`. Neither recursion nor containers (lists, dictionaries, sets, etc) are allowed in your solution.

```
def make_zipper(f1, f2, sequence):
    """ Return a function of f1 and f2 composed based on sequence.

    >>> def increment(x):
            return x + 1
    >>> def square(x):
            return x * x
    >>> do_nothing = make_zipper(increment, square, 0)
    >>> do_nothing(2)      # Don't call either f1 or f2, just return your input untouched
    2
    >>> incincsq = make_zipper(increment, square, 112)
    >>> incincsq(2)        # increment(increment(square(2))), so 2 → 4 → 5 → 6
    6
    >>> sqincsqinc = make_zipper(increment, square, 2121)
    >>> sqincsqinc(2)      # square(increment(square(increment(2)))), so 2 → 3 → 9 → 10 → 100
    100
    """


    zipper = _____

    helper = _____

    while _____:

        if _____ == 1:

            zipper = helper(f1, _____ )

        else:

            zipper = helper(f2, _____ )

        sequence = _____

    return zipper
```