CS 61A
Fall 2020

Structure and Interpretation of Computer Programs

Quiz 1 Solutions

INSTRUCTIONS

- Please review this worksheet before the exam prep session. Coming prepared will help greatly, as the TA will be live solving without allocating much time for individual work.
- Either Sean or Derek will be on video live solving these questions. The other TA will be answering questions in the chat. It is in your best interest to come prepared with **specific** questions.
- This is not graded, and you do not need to turn this in to anyone.
- Fall 2020 students: the boxes below are an artifact from more typical semesters to simulate exam environments. Obviously this doesn't apply to this semester's exams, but we just kept the fields to keep our materials looking professional :) Feel free to ignore them.
- For multiple choice questions, fill in each option or choice completely.
  - ☐ means mark **all options** that apply
  - ◯ means mark a **single choice**

Last name	
First name	
Student ID number	
CalCentral email (_@berkeley.edu)	
Discussion Section	___ ___ ___
All the work on this exam is my own. **(please sign)**	

1. **My Last Three Brain Cells**

   (a) A `k`-*memory function* takes in a single input, prints whether that input was seen exactly `k` function calls ago, and returns a new `k`-memory function. For example, a 2-memory function will display "Found" if its input was seen exactly two function calls ago, and otherwise will display "Not found".

   Implement `three_memory`, which is a three-memory function. You may assume that the value `None` is never given as an input to your function, and that in the first two function calls the function will display "Not found" for any valid inputs given.

```
def three_memory(n):
    """
    >>> f = two_memory('first')
    >>> f = f('first')
    Not found
    >>> f = f('second')
    Not found
    >>> f = f('third')
    Not found
    >>> f = f('second')
    Not found
    >>> f = f('second')
    Found
    >>> f = f('third')
    Found
    """
    def f(x, y, z):
        def g(i):
            if i == x:
                print('Found')
            else:
                print('Not found')
            return f(y, z, i)
        return g
    return f(None, None, n)
```

From the skeleton, we know that `three_memory` will return whatever `f` returns. Because we want `three_memory` to return a three-memory function, then, we know that `f` should also return a three-memory function. Furthermore, we see that `f` takes in three arguments; we have to keep track of three past values at a time (the last three values that our three-memory functions have been called on) to keep track of whether to print "Found" or "Not found" upon future calls.

Because `n` is the first value fed into the function and it is placed into the parameter `z`, we can infer that `z` contains the value that was seen exactly one function call ago. We can then let `y` contain the value that was seen exactly two function calls ago, and `x` contain the value that was seen exactly three function calls ago. `f` will then return `g`; the reasoning behind this is that our three-memory function has to have access to `x`, `y` and `z` in order to compare future inputs against values that have been seen in the past. `g` is a function of one variable that is a child function of `f`, so it has everything we need.

Within `g`, we check to see if the new input `i` is equal to `x`, the value seen exactly three function calls ago, in order to decide what to print. Then we call `f` again to shift all of the parameters over. `y`, which was seen two function calls ago, has now been seen 3 function calls ago after this call and so we shift it to `x`. Similarly, `x` is shifted to `y`, and `i` is passed to `z`. The previous `x` value is discarded, because we don't have to keep track of what was seen four or more calls ago.

For a more in-depth explanation, see the recording of exam-prep section.

2. **Welcome to CS 61A!**

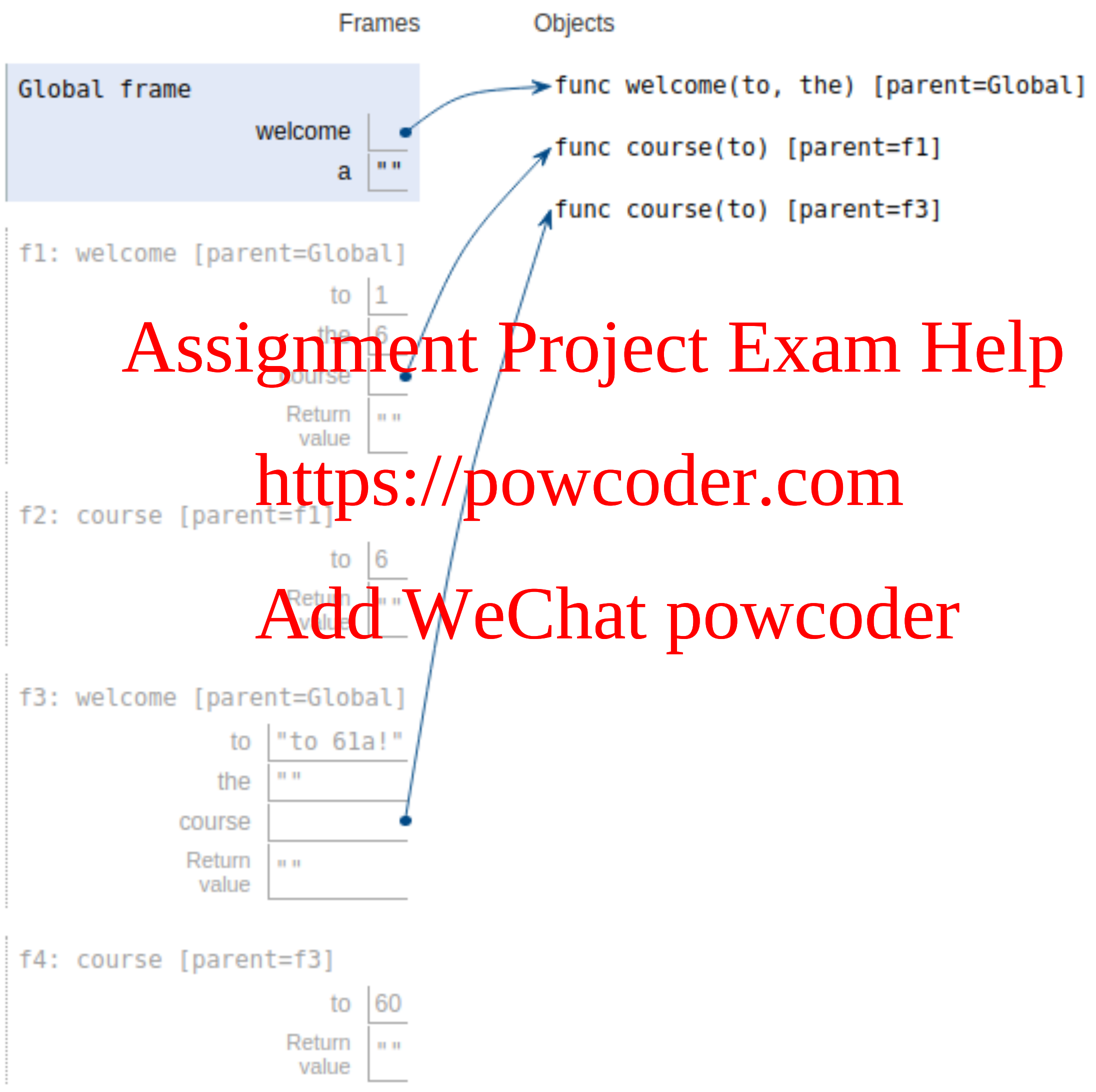
Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* A complete answer will:

(a) Add all missing names and parent annotations to all local frames.

(b) Add all missing values created or referenced during execution.

(c) Show the return value for each local frame.

You must list all bindings in the order they first appear in the frame.



3. **Run, K, Run**

   (a) An *increasing run* of an integer is a sequence of consecutive digits in which each digit is larger than the last. For example, the number 12344345 has four increasing runs: 1234, 4, 4 and 345. Each run can be indexed **from the end** of the number, starting with index 0. In the previous example, the 0th run is 345, the first run is 4, the second run is 4 and the third run is 1234.

   Implement `get_k_run_starter`, which takes in integers `n` and `k` and returns the 0th digit of the `k`th increasing run within `n`. The 0th digit is the leftmost number in the run. You may assume that there are at least `k+1` increasing runs in `n`.

```
def get_k_run_starter(n, k):
    """
    >>> get_k_run_starter(123412341234, 1)
    1
    >>> get_k_run_starter(1234234534564567, 0)
    4
    >>> get_k_run_starter(1234234534564567, 1)
    3
    >>> get_k_run_starter(1234234534564567, 2)
    2
    """
    i = 0

    final = None

    while i <= k:

        while n > 10 and (n % 10 > (n//10) % 10):

            n = n // 10

        final = n % 10

        i += 1

        n //= 10

    return final
```

`i` will count which run we are currently processing. Each run through the outer `while` loop will process exactly one run by repeatedly taking digits off of the end of `n` until it gets to the boundary between runs, as seen in the condition of the inner `while` loop. After we reach the boundary between two runs, we save the leftmost digit of the run just processed into the variable `final`. Eventually, once we process the `k`th run the leftmost digit of the `k`th run will be stored in `final`, the outer `while` loop will quit and we will return `final`.

For a more in-depth solution and derivation of how to get there, see the recording of the exam prep section.

4. **It's Always a Good Prime**

(a) Implement `div_by_primes_under`, which takes in an integer `n` and returns an `n`-divisibility checker. An *n-divisibility-checker* is a function that takes in an integer `k` and returns whether `k` is divisible by any integers between 2 and `n`, inclusive. Equivalently, it returns whether `k` is divisible by any primes less than or equal to `n`.

```
def div_by_primes_under(n):
    """
    >>> div_by_primes_under(10)(11)
    False
    >>> div_by_primes_under(10)(121)
    False
    >>> div_by_primes_under(10)(12)
    True
    >>> div_by_primes_under(5)(1)
    False
    """
    checker = lambda x: False

    i = 2

    while i <= n:

        if not checker(i):

            checker = (lambda f, i: lambda x: x % i == 0 or f(x))(checker, i)

        i += 1

    return checker
```

Because we want to test if a future input `k` is divisible by any integers between `2` and `n` inclusive, it makes sense to progress through each of these values one at a time and use them somehow. So `i` begins at 2 and is incremented at each step until it is greater than `n`, at which point the loop breaks.

From the skeleton, we know that `checker` must be a one-argument function which returns something with a meaningful truth value. So we may conjecture that `checker` is the function that we want to return; because we're updating it on some loops, it must capture some useful information from the `i` values we progress through. If we assume that it correctly returns `True` if `i` is divisible by any numbers it has seen so far (that is, between `2` and `i-1`, inclusive) then it becomes clear that the if statement is only entered if `i` is prime – by definition, no integers between `2` and `i-1` divide it evenly.

So each time we hit a prime `i`, we should update `checker` so that it checks whether future values are divisible by `i`. A first attempt might be to write `checker = lambda x: x % i == 0 or checker(x)`, so that the function returns true if its input is divisible by `i` or if it is divisible by anything bewteen `2` and `i-1`, inclusive, which is what `checker` provides. But when this function is called it doesn't have `i` or `checker` in its local frame and so will grab these values from the parent frame; this is at the very end of the function call, and so it will reference the wrong function. To "freeze" the values of `checker` and `i` within the lambda function, we want to define a new frame with local variables holding them. To do so, we define a lambda function taking in `f` and `i` and returning the new checker function, then immediately call that outer lambda function on `checker` and `i` to save those values for later.

For a more in-depth explanation, see the recording of exam-prep section.

5. **Appealing Apple-Peeling App**

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:

```
def apples(to, apples):
    while apples > 0 and to > 0:
        apples, to = to, apples
        apples -= 1
    return to

def banana(split):
    return lambda: split

onion = lambda z: lambda x: lambda y: x(y(z))
slide = lambda x: x(x)
```

Expression	**Interactive Output**
`2 + 3`	`5`

`print(print(print(2, 3), 4) + 5)`	`2 3`
	`None 4`
	`Error`

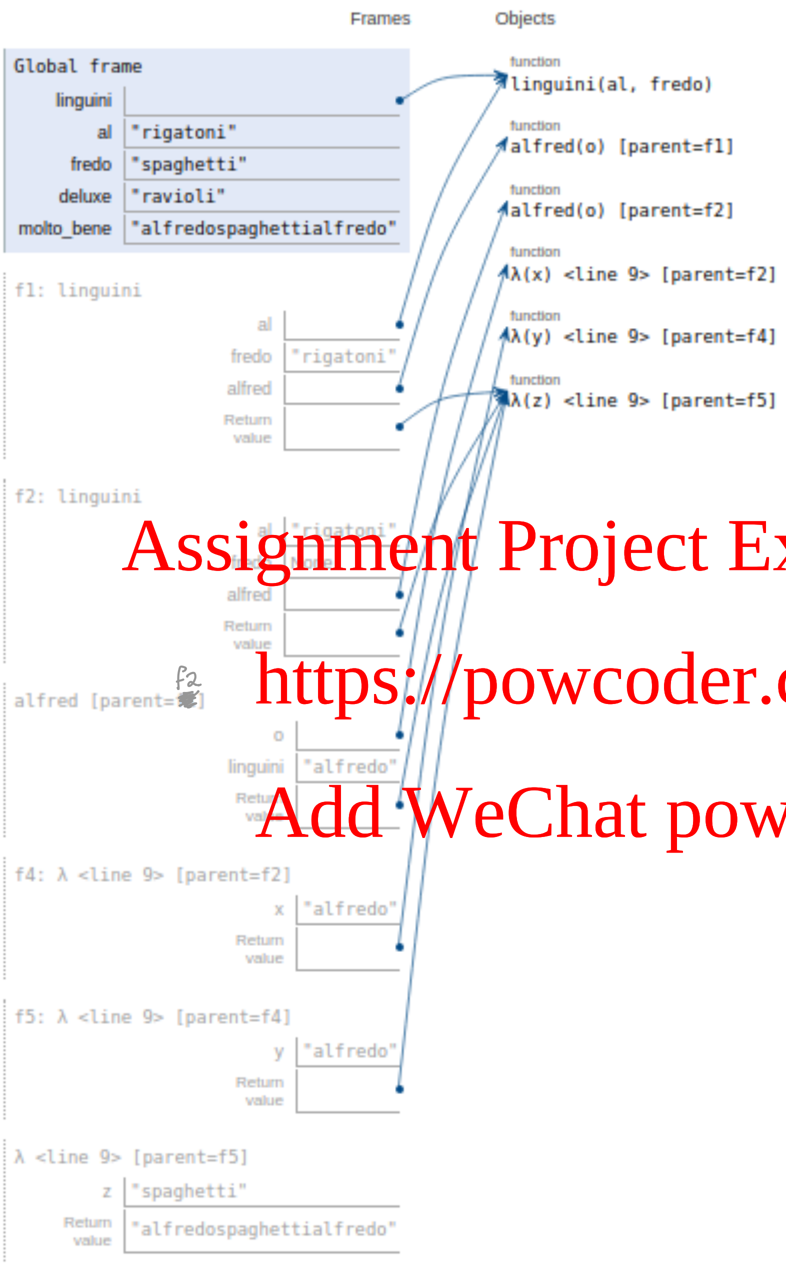
`print(apples(10, 9), apples(10, 10), apples(10, 11))`	`1 1 2`
`banana(3)`	`Function`
`onion(3)(banana)(banana)()()`	`3`
`onion(banana)(banana)(3)()()`	`Error`

`f = slide(onion)`
`f(lambda x: x)(lambda x: 2*x)(3)`
	Error

`f = slide(onion)`
`f(lambda x: x)(lambda x: x)(3)(lambda x: 2*x)(lambda x: 5*x)`
	`30`

6. **Im-pasta-ble**

For each blank in the code, choose any appropriate value which would cause it to produce the given environment diagram.


Frames
Objects
Global frame
linguini
al "rigatoni"
fredo "spaghetti"
deluxe "ravioli"
molto_bene "alfredospaghettialfredo"
f1: linguini
al
fredo "rigatoni"
alfred
Return value
f2: linguini
al "rigatoni"
fredo None
alfred
Return value
alfred [parent=f2]
o
linguini "alfredo"
Return value
f4: λ <line 9> [parent=f2]
x "alfredo"
Return value
f5: λ <line 9> [parent=f4]
y "alfredo"
Return value
λ <line 9> [parent=f5]
z "spaghetti"
Return value "alfredospaghettialfredo"
function linguini(al, fredo)
function alfred(o) [parent=f1]
function alfred(o) [parent=f2]
function λ(x) <line 9> [parent=f2]
function λ(y) <line 9> [parent=f4]
function λ(z) <line 9> [parent=f5]

```
def linguini(al, fredo):
    if not al:
        return 'fredo'
    def alfred(o):
        linguini = 'alfredo'
        return o(linguini)(linguini)
    if fredo:
        return al(fredo, None)
    return alfred(lambda x: lambda y: lambda z: x+z+y)

al, fredo, deluxe = 'rigatoni', 'spaghetti', 'ravioli'
molto_bene = linguini(linguini, al)(fredo)
```

(a) **First blank**
A. `not al` B. `not fredo` C. `al != fredo` D. `fredo` E. `linguini`

(b) **Second blank**
A. `linguini` B. `alfred` C. `o` D. `fredo` E. `al`

(c) **Third blank**
A. `alfredo` B. `o` C. `linguini` D. `fredo` E. `al`

(d) **Fourth blank**

A. `alfredo` B. `o` C. `linguini` D. `fredo` E. `al`

(e) **Fifth blank**
A. `alfred`

B. `lambda x: x`
C. `lambda x: lambda y: x + y`
D. `lambda x: lambda y: lambda z: y + x + z`

E. `lambda x: lambda y: lambda z: x + z + y`

(f) **Sixth blank**
A. `al` B. `fredo` C. `deluxe` D. `linguini` E. `None`

(g) **Seventh blank**
A. `al` B. `fredo` C. `deluxe` D. `linguini` E. `None`

A, C, C, C, E, A, B