CS 61A Spring 2020

Structure and Interpretation of Computer Programs

Final Exam

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, on Google Forms. If either tool stops working, switch to the other one and continue taking the exam. We will merge your solutions together at the end of the exam, taking Google Form submissions in preference to submissions at exam.cs61a.org.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

◯ You must choose either this option

◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

You can complete and submit these questions before the exam starts.

(a) What is your full name?

(b) What is your student ID number?

(c) What is your @berkeley.edu email address?

(d) Who is your TA? (See cs61a.org/staff.html for pictures.)

1. (8 points) What Does This Function Do?

Complete the description of each function so that it correctly describes the function's behavior.

(a) (4 points)

```
def count(n, t, k):
    if n == 0:
        return int(t <= 0)
    elif k > n:
        return 0
    else:
        a = count(n, t, k + 1)
        b = count(n-k, t-1, k)
        return a + b
```

Hint: `int(False)` evaluates to `0` and `int(True)` evaluates to `1`.

As in `count_partitions` from the Midterm 2 Study Guide, a "way of summing to `n` using parts" is a sum of zero or more positive integers (the parts) that appear in non-decreasing order and total `n`. For example, `1 + 2` is a way of summing to `3` using 2 parts, but `2 + 1` is not.

Complete this description: `count(n, t, k)` counts the ways of summing to `n` using ...

i. (2 pt)

- ◯ ... at least `t` parts ...
- ◯ ... at most `t` parts ...
- ◯ ... at least `k` parts ...
- ◯ ... at most `k` parts ...

ii. (2 pt)

- ◯ ... that are all less than or equal to `t`.
- ◯ ... that are all greater than or equal to `t`.
- ◯ ... that are all less than or equal to `k`.
- ◯ ... that are all greater than or equal to `k`.

(b) (4 points)

```
def prime(n):
    """Return the smallest prime number larger than n.

    >>> prime(2)
    3
    >>> prime(8)
    11
    >>> prime(prime(8))
    13
    >>> prime(prime(prime(8)))
    17
    """
    <implementation omitted>

def again():
    f, g = prime, prime
    def h(x):
        nonlocal g
        g, h = (lambda h: lambda y: h(f(y)))(g), g(x)
        return h
    return h
```

Assume that `prime` is implemented correctly and behaves as its docstring describes.

Below, *applying `prime` to `x` repeatedly 3 times* means evaluating `prime(prime(prime(x)))`.

Complete this description: `again()` returns a function `h` that takes a number `x` and returns the result of applying `prime` to `x` repeatedly . . .

i. (2 pt)

- ◯ . . . `k-1` times . . .
- ◯ . . . `k` times . . .
- ◯ . . . `k+1` times . . .
- ◯ . . . `2 ** k` times . . .

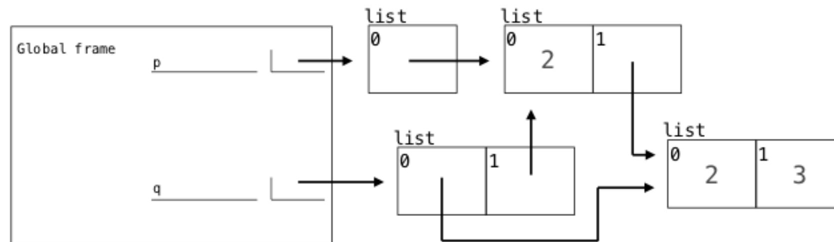
ii. (2 pt)

- ◯ . . . where `k` is the number of times that this `h` function has been called.
- ◯ . . . where `k` is the number of times that `prime` has been called during a call to this `h` function.
- ◯ . . . where `k` is the total number of times that all `h` functions returned by calling `again()` (perhaps multiple times) have been called throughout the whole program.
- ◯ . . . where `k` is the total number of times that `prime` has been called (perhaps in other ways than by this `h` function) throughout the whole program.

2. (12 points) Mind Your P's and Q's

(a) (6 points)

Fill in each blank in the code example below so that its environment diagram is the following:



https://i.imgur.com/xPJsDGg.png

```
p = [[2], [2, 2]]

p[0].________(_________)
         (a)        (b)

q = [_________, _________]
         (c)          (d)

p_________ = 3
    (e)
```

i. (1 pt) Which of the following names could complete blank (a)?

◯ `add`

◯ `pop`

◯ `append`

◯ `extend`

ii. (1 pt) Which of the following expressions could complete blank (b)?

◯ `p`

◯ `p[0]`

◯ `p[1]`

◯ `p[:]`

iii. (1 pt) Which of the following expressions could complete blank (c)?

◯ `p.pop()`

◯ `p[1]`

◯ `p[0]`

◯ `p[:1]`

iv. (1 pt) Which of the following expressions could complete blank (d)?

◯ `p.pop()`

◯ `p[1]`

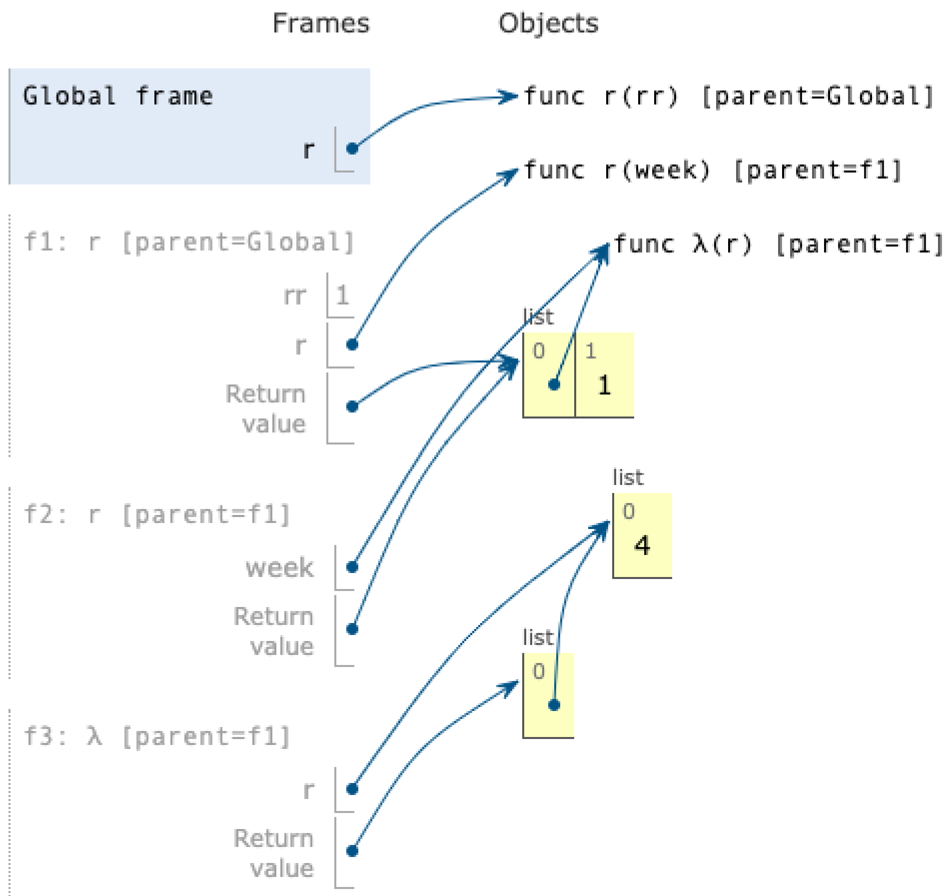
◯ `p[0]`

◯ `p[:1]`

v. (2 pt) Write code that could complete blank (e).

(b) (6 points)

Fill in each blank in the code example below so that its environment diagram is the following:



https://i.imgur.com/qk83xRw.png

```
def r(rr):

    if rr:

        def r(week):

            return [_________, rr]
                       (a)

    rr = _________
            (b)

    return r(_________)
                (c)

r(5)_________
       (d)
```

Note: Line numbers for lambda functions have been omitted intentionally.

i. (1 pt) Write an expression that could complete blank (a).

ii. (1 pt) Write an expression that could complete blank (b).

iii. (2 pt) Write an expression that could complete blank (c).

iv. (2 pt) Which of these could complete blank (d)?

- ○ `.pop()([4])`
- ○ `.pop()(4)`
- ○ `[0](4)`
- ○ `[0]([4])`

3. (12 points) Bounds

Definitions: A *bound* is a two-element tuple of numbers in which element 0 is smaller than element 1. A number `t` is *contained in* bound `b` if `b[0] < t and t < b[1]`. How *tight* a bound `b` is around a number `t` describes the largest absolute distance between `t` and one of the numbers in `b`. For example, the tightness of bound `(1, 7)` around `6` is 5 because the absolute difference between 6 and 1 is 5.

(a) (2 points)

Implement `minimum`, which takes a list `s` and a one-argument function `key`. It returns the value in `s` for which `key` produces the smallest return value. If `s` is empty, `minimum` returns `None`. If more than one value in `s` produces a `key` value at least as small as all others, then `minimum` returns the first.

```
def minimum(s, key):
    """Return the first value v in s for which key(v) is less than or equal to
    key(w) for all values w in s. Return None if s is empty.

    >>> minimum([5, 4, 3, 2, 1], lambda x: abs(x - 3.1)) # Closest to 3.1
    3
    >>> a = [3]
    >>> minimum([[5], [4], a, [3], [2], [1]], lambda x: abs(x[0] - 3.1)) is a
    True
    """
    if not s:

        return None

    m = s[0]

    for v in s[1:]:

        if _________:
               (a)

            m = v

    return m
```

i. (2 pt) What expression completes blank (a)?

Important: You may not call the built-in `min` or `max` functions for this blank.

(b) (4 points)

Implement `tightest`, which takes a list of `bounds` and a number `t`. It returns the first bound in `bounds` that both contains `t` and is the most tight around `t`. If no bound in `bounds` contains `t`, `tightest` returns `None`.

Assume `minimum` is implemented correctly.

```
def tightest(bounds, t):
    """Return the tightest bound around t in bounds.

    >>> bounds = [(2, 6), (3, 4), (1, 5), (1, 6), (0, 4)]
    >>> tightest(bounds, 3)
    (1, 5)
    >>> tightest(bounds, 3.1)
    (3, 4)
    >>> tightest(bounds, 5)
    (2, 6)
    >>> tightest(bounds, 2)
    (0, 4)
    >>> print(tightest(bounds, 6))
    None
    """
    return minimum([b for b in bounds if _________],
                                                (a)

                   lambda b: _________)
                                (b)
```

i. (2 pt) What expression completes blank (a)?

Important: You may not call the built-in `min` or `max` functions for this blank.

ii. (2 pt) What expression completes blank (b)?

- ◯ `max(t - b[0], b[1] - t)`
- ◯ `abs(t - max(b))`
- ◯ `[abs(t - x) for x in b][0]`
- ◯ `abs(max([t - x for x in b]))`

(c) (6 points)

Implement `nest`, which takes a list of `bounds`. It returns the largest number of bounds in the list that all overlap with each other.

```
def overlap(a, b):
    """Return whether there is some number t contained in both a and b.

    >>> overlap([2, 4], [1, 3])  # 2.5 is contained in both bounds.
    True
    >>> overlap([1, 3], [2, 4])  # 2.5 is contained in both bounds.
    True
    >>> overlap([2, 4], [1, 2])  # No number is contained in both bounds.
    False
    """
    return a[0] < b[1] and b[0] < a[1]

def nest(bounds):
    """Return the maximum number of bounds that all contain the same number.

    >>> bounds = [(2, 6), (3, 4), (1, 5), (1, 6), (0, 4), (0, 3)]
    >>> nest(bounds)  # All but the last contain 3.1, so these 5 all overlap with each other.
    5
    >>> bounds = [(1, 5), (5, 7), (7, 9), (1, 9)]
    >>> nest(bounds)  # Any of the first three overlaps with the last, but not with each other.
    2
    >>> bounds = [(1, 9), (1, 5), (5, 7), (7, 9)]
    >>> nest(bounds)  # The first overlaps with any of the last three.
    2
    >>> bounds = [(2, 4), (1, 3), (1, 2)]
    >>> nest(bounds)  # Any two consecutive bounds overlap, but the first & last do not overlap.
    2
    """
    if not bounds:

        return 0

    rest = [b for b in bounds[1:] if overlap(b, _________)]
                                                   (a)

    return max(nest(_________), 1 + _________)
                        (b)            (c)
```

i. (2 pt) What expression completes blank (a)?

ii. (2 pt) What expression completes blank (b)?

- ◯ `bounds`
- ◯ `bounds[1:]`
- ◯ `rest`
- ◯ `bounds[0] + rest`

iii. (2 pt) What expression completes blank (c)?

No more questions.