CS 61A Structure and Interpretation of Computer Programs
Fall 2022

Midterm 2 Solutions

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

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?

1. (7.0 points) What would Python Python?

Assume the code below has been executed.

```
class Snake:
    legs = 0
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name
    def run(self, s):
        print("Snakes don't run")
        return self.crawl()
    def crawl(self):
        print(f"{self} crawled")
    def eat(self, s):
        self.run(s)
        print("Nom nom")

class Python(Snake):
    def run(self, s):
        print(eval(s))  # eval(s) evaluates string s as a Python expression

snek = Snake("atari")
solidsnake = Snake("David")
solidsnake.legs = 2
solidsnake.run = lambda s: print("He ran")
python = Python("pypy")
```

For each expression below, write the output **displayed by the interactive Python interpreter** when the expression is evaluated.

- If an output has multiple lines, write each line separately.
- 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".

Each expression below should be evaluated independently of previously-evaluated expressions.

(a) (1.0 pt) `[snek.legs, Snake.legs]`

[0, 0]

(b) (2.0 pt) `solidsnake.eat("python")`

He ran
Nom nom

(c) (2.0 pt) `python.eat("snek")`

atari
Nom nom

(d) (2.0 pt) `Snake.run(python, python)`

Snakes don't run
pypy crawled

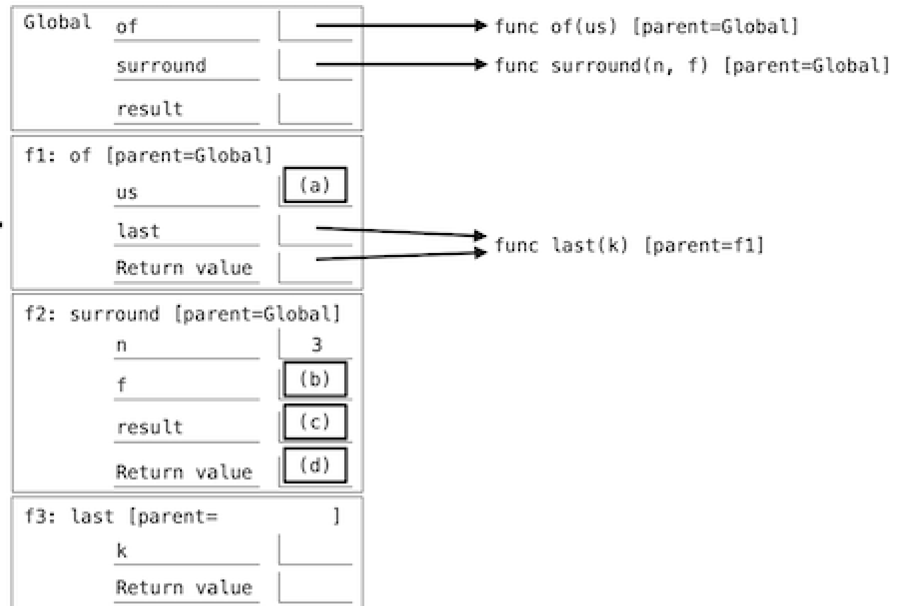
2. (6.0 points) Environmental Disaster

Your friend asks you to help debug their program below, which doesn't print what they expect. Complete the environment diagram, then answer questions about it. The diagram itself will not be scored; only the questions based on it.

```
def of(us):
    def last(k):
        "The last k items of us"
        while k > 0:
            result.append(us.pop())
            k = k - 1
        return result
    return last

def surround(n, f):
    "n is the first and last item of f(2)"
    result = [n]
    result = f(2)
    result[0] = [n]
    return result.append(n)

result = [1]
surround(3, of([4, 5, 6]))
print(result)
```



(a) (1.0 pt) *Blank (a)*: What is the name `us` bound to in f1?

- ◯ A list `[4, 5, 6]`
- ◯ A list `[4, 5]`
- ● A list `[4]`

(b) (1.0 pt) *Blank (b)*: What is the name `f` bound to in frame `f2`?

- ◯ `func of(us) [parent=Global]`
- ● `func last(k) [parent=f1]`
- ◯ None of the above

(c) (1.0 pt) *Blank (c)*: What is the name `result` bound to in frame `f2`?

- ● The same list bound to the name `result` in the Global frame
- ◯ A list that is **not** bound to any name in the Global frame

(d) (1.0 pt) *Blank (d)*: What is the return value of frame `f2`?

- ◯ The same list bound to the name `result` in `f2`
- ◯ A list that is one element longer than the one bound to the name `result` in `f2`
- ● None of the above

(e) (2.0 pt) What is displayed by the last line, `print(result)`?

```
[[3], 6, 5, 3]
```

3. (8.0 points) Hog Revisited

In Project 1, we created a simulator for the game of Hog without using lists or iterators. Let's revise the implementation, but instead of representing dice as zero-argument functions, **we'll represent dice as iterators over outcomes**.

(a) (3.0 points)

Implement `make_test_dice`, which takes a list of `outcomes`. It returns an infinite iterator that repeatedly cycles through the outcomes.

```
def make_test_dice(outcomes):
    """Return an infinite iterator that cycles through the elements of outcomes.

    >>> dice = make_test_dice([1, 2, 3])
    >>> next(dice)
    1
    >>> next(dice)
    2
    >>> next(dice)
    3
    >>> next(dice)
    1
    >>> next(dice)
    2
    >>> next(dice)
    3
    >>> next(dice)
    1
    >>> next(dice)
    2
    """
    outcomes = list(outcomes)
    while True:
        for i in ___(a)___:
            ___(b)___
```

i. (1.0 pt) Which of these could fill in blank (a)? Select **all** that apply.

- ■ `outcomes`
- ☐ `outcomes + make_test_dice(outcomes)`
- ☐ `outcomes.extend(make_test_dice(outcomes))`
- ☐ `outcomes[0] + make_test_dice(outcomes[1:])`
- ☐ `make_test_dice(outcomes) + make_test_dice(outcomes)`
- ☐ `make_test_dice(outcomes[0]) + make_test_dice(outcomes[1:])`

ii. (2.0 pt) Fill in blank (b).

```
yield i
```

(b) (5.0 points)

Implement `roll_dice`, which takes two arguments: a positive integer called `num_rolls` giving the number of dice to roll and a `dice` iterator that provides outcomes. It returns the number of points scored by rolling the `dice` `num_rolls` times in a turn. The points scored are either the sum of the dice outcomes or 1 (Sow Sad).

- Sow Sad. If any of the dice outcomes is a 1, the current player's score for the turn is 1.

```
def roll_dice(num_rolls, dice):
    """Return the number of points scored by rolling dice num_rolls times.

    >>> dice = make_test_dice([6, 6, 6, 6, 6, 1])
    >>> roll_dice(5, dice) # 6, 6, 6, 6, 6
    30
    >>> roll_dice(5, dice) # 1, 6, 6, 6, 6
    1
    >>> roll_dice(2, dice) # 6, 1
    1
    >>> roll_dice(10,  make_test_dice([2, 4, 3])) # 2, 4, 3, 2, 4, 3, 2, 4, 3, 2
    29
    """
    rolls = ___(a)___
    if ___(b)___:
        return 1
    ___(c)___
```

i. (2.0 pt) Fill in blank (a).

```
[next(dice) for _ in range(num_rolls)]
```

ii. (1.0 pt) Fill in blank (b).

```
1 in rolls
```

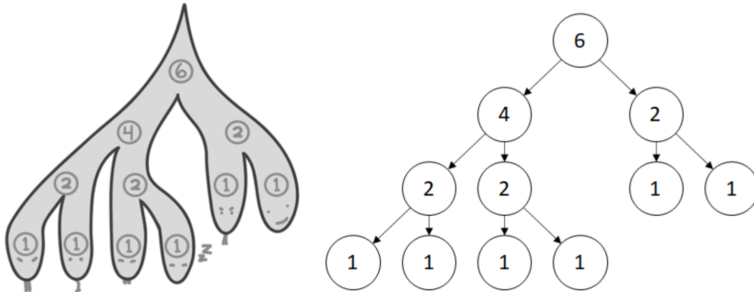
iii. (2.0 pt) Fill in blank (c).

```
return sum(rolls)
```

4. (9.0 points) The Lambdanean Hydra

Heracles is fighting the Lambdanean Hydra, a many-headed monster, where two heads regrow when one head gets chopped off. A hydra is represented by a tree, each head is represented by a leaf labeled 1, and each non-leaf node has two branches and is labeled with the number of leaves among its descendants. Here's a six-headed hydra named `lerna`:

```
lerna=Tree(6,[Tree(4,[Tree(2,[Tree(1),Tree(1)]),Tree(2,[Tree(1),Tree(1)])]),Tree(2,[Tree(1),Tree(1)])]
```



(a) (4.0 points)

Implement `is_hydra`, which takes in as input a Tree instance `t` and determines if `t` represents a valid hydra.

```
def is_hydra(t):
    """Return True if Tree instance t properly represents a hydra and False otherwise.

    >>> is_hydra(Tree(1))
    True
    >>> is_hydra(Tree(2, [Tree(1), Tree(1)]))
    True
    >>> is_hydra(Tree(3, [Tree(1), Tree(2)]))  # Wrong leaf label
    False
    >>> is_hydra(Tree(3, [Tree(1), Tree(1)]))  # Wrong root label
    False
    >>> is_hydra(Tree(3, [Tree(3, [Tree(1), Tree(1)]), Tree(1)]))  # Wrong node label below root
    False
    >>>is_hydra(lerna)  # lerna the six-headed hydra is defined above
    True
    """
    if t.is_leaf():
        return t.label == 1
    if len(t.branches) != 2:
        return False
    return (t.label == ___(a)___) and ___(b)___
```

i. (2.0 pt) Fill in blank (a).

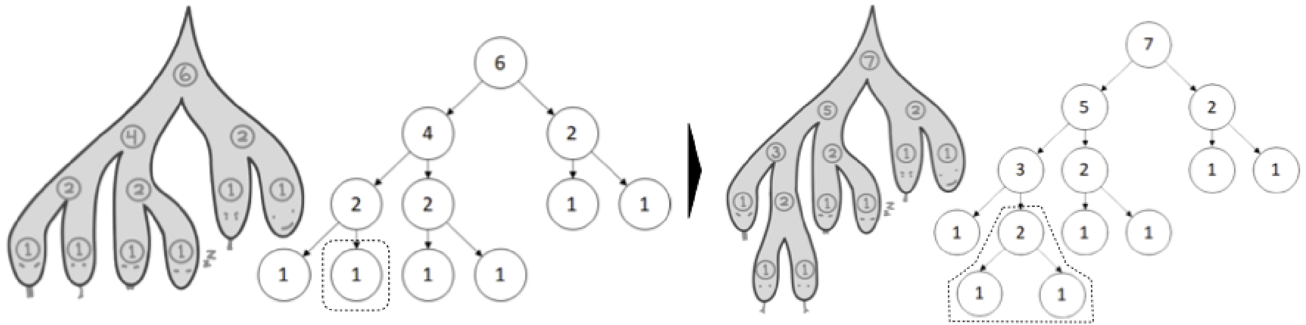
```
sum([b.label for b in t.branches])
```

ii. (2.0 pt) Fill in blank (b).

```
is_hydra(t.branches[0]) and is_hydra(t.branches[1])
```

(b) (5.0 points)

If Heracles cuts off the second head from the left, two new heads appear there, and all ancestor labels are updated.



Implement `chop_head(hydra, n)`, which takes as input a Tree instance `hydra` representing a hydra and a positive integer `n`. It mutates `hydra` by chopping off the `n`th head from the left (adding two new adjacent heads in its place).

```
def chop_head(hydra, n):
    """
    >>> lerna = Tree(1)
    >>> chop_head(lerna, 1)  # Note that n is 1-indexed
    >>> chop_head(lerna, 1)
    >>> chop_head(lerna, 3)
    >>> chop_head(lerna, 1)
    >>> chop_head(lerna, 3)
    >>> lerna                # Now lerna is a six-headed hydra (above left)
    Tree(6,[Tree(4,[Tree(2,[Tree(1),Tree(1)]),Tree(2,[Tree(1),Tree(1)])]),
            Tree(2,[Tree(1),Tree(1)])])
    >>> chop_head(lerna, 2)
    >>> lerna                # The mutated lerna now has seven heads (above right)
    Tree(7,[Tree(5,[Tree(3,[Tree(1),Tree(2,[Tree(1),Tree(1)])]),Tree(2,[Tree(1),Tree(1)])]),
            Tree(2,[Tree(1),Tree(1)])])
    """
    assert is_hydra(hydra)
    assert n > 0 and n <= hydra.label
    if hydra.is_leaf():
        ___(a)___  # This blank may be filled with multiple lines of code
        return
    ___(b)___
    left, right = hydra.branches
    if ___(c)___:
        chop_head(right, ___(d)___)
    else:
        chop_head(left, n)
```

i. (2.0 pt) Select **all** options that could fit in blank (a), which may have multiple lines.

```
Option 1:  hydra = Tree(2,[Tree(1), Tree(1)])

Option 2:  for _ in range(2):
               hydra.branches.append(Tree(1))
               hydra.label = 2

Option 3:  hydra.label = Tree(2)
           hydra.branches = [Tree(1),Tree(1)]

Option 4:  hydra.label = 2
           hydra.branches = [Tree(1), Tree(1)]

Option 5:  hydra.label = 2
           hydra.branches = [Tree(1)] * 2
```

- ☐ Option 1
- ■ Option 2
- ☐ Option 3
- ■ Option 4
- ☐ Option 5

ii. (1.0 pt) Fill in blank (b).

```
hydra.label += 1
```

iii. (1.0 pt) Fill in blank (c).

```
n > left.label
```

iv. (1.0 pt) Fill in blank (d).

```
n - left.label
```

5. (7.0 points) Aim for 100

The function `count_subsets` takes as input a list of positive integers `s`. It returns the number of lists that sum to 100 and contain a subset of the elements of `s` in order.

```
def count_subsets(s):
    """
    >>> count_subsets([25, 50, 75, 100, 125, 150])  # [25, 75], [100]
    2
    >>> count_subsets([25, 50, 25, 75])  # [25, 75] (first 25), [25, 75] (second 25), [25, 50, 25]
    3
    >>> count_subsets(list(range(1,10000)))
    444793
    """
```

(a) (5.0 points)

Complete the following implementation of `count_subsets`.

```
def count_subsets(s):
    def helper(sum_so_far, index):
        if ___(a)___:
            if ___(b)___:
                return 1
            return 0
        return ___(c)___ + ___(d)___
    return helper(0,0)
```

i. (1.0 pt) Fill in blank (a).

- ○ `index == s`
- ● `index == len(s)`
- ○ `sum_so_far == 100`
- ○ `sum_so_far != 100`

ii. (1.0 pt) Fill in blank (b).

- ○ `index == s`
- ○ `index == len(s)`
- ● `sum_so_far == 100`
- ○ `sum_so_far != 100`

iii. (1.0 pt) Fill in blank (c).

- ○ `count_subsets(s[index:])`
- ○ `count_subsets(s[1:])`
- ○ `helper(sum_so_far, index)`
- ● `helper(sum_so_far, index + 1)`

iv. (2.0 pt) Fill in blank (d).

```
helper(sum_so_far+s[index], index + 1)
```

(b) (2.0 points)

i. (1.0 pt) What is the order of growth of the time required to evaluate `count_subsets`, in terms of the length of `s`?

- ● Exponential
- ○ Quadratic
- ○ Linear
- ○ Logarithmic
- ○ Constant

ii. (1.0 pt) We decide to rewrite `count_subsets`, following a different approach:

```
def count_subsets(s):
    values = [1]+[0]*100
    for i in s:
        for j in reversed(range(100-i+1)):
            values[j+i]+=values[j]
    return values[100]
```

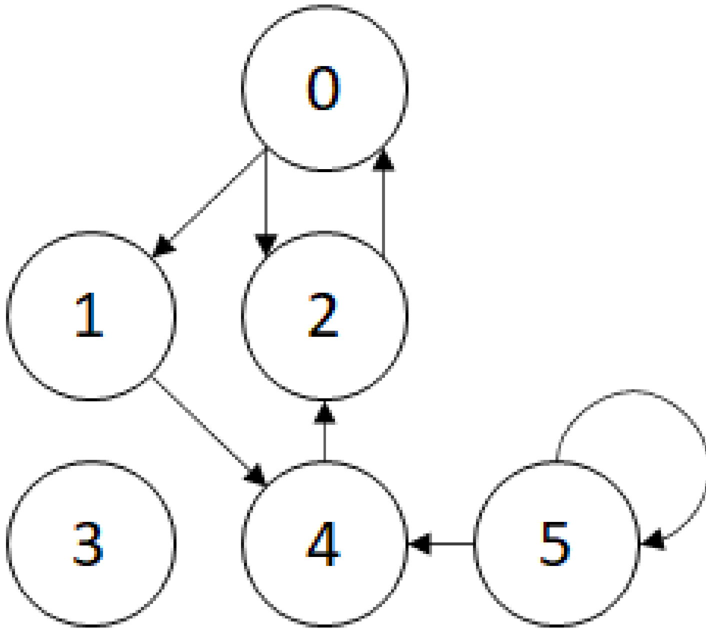
What is the order of growth of the time required to evaluate the new version of `count_subsets`, in terms of the length of `s`?

- ○ Exponential
- ○ Quadratic
- ● Linear
- ○ Logarithmic
- ○ Constant

6. (8.0 points) Point A to Point B

A *graph* is a structure consisting of two parts:

- A set of n nodes, which are labeled from 0 to n-1.
- A set of directed edges, which go from one node (the source node) to another node (the destination).



The above is an example of a graph with 6 nodes, with edges represented as arrows. Note that a graph can have:

- Nodes that have no edges to/from them (Node 3)
- Two edges between the same two nodes, with source and destination reversed (The edges between Nodes 0 and 2)
- An edge from a node to itself (Node 5)

However, a graph cannot have two distinct edges from one node to another. For example, we cannot have a graph where two edges go from Node 0 to Node 1.

We say that a Node j is a **destination** of Node i if there is a directed edge from Node i to Node j. Thus, in the above example, Node 0 has two destinations (Nodes 1 and 2), and Node 1 has one destination (Node 4). Note that Node 0 is not considered a destination of Node 1.

We say that there is a **path** from Node i to Node j if either `i==j`, or we can go from Node i to Node j by following a sequence of edges in the graph. For example, we can get from Node 1 to Node 0 by following the edges `1->4`, then `4->2`, then `2->0`.

Graphs are often used to represent transportation maps, where nodes represent individual locations, and edges represent a path you can take to go from one node to another.

A `Graph` instance is constructed from a number of nodes and begins with no edges. Edges are added, and then the `destinations` method lists destinations for a node and the `has_path` method determines if a path exists between two nodes.

```
>>> g = Graph(6)      # A graph with 6 nodes and no edges.
>>> g.add_edge(0, 2)  # Add an edge from node 0 (source) to node 2 (destination)
>>> g.add_edge(2, 0)  # Add an edge from node 2 (source) to node 0 (destination)
>>> g.add_edge(0, 1)
>>> g.add_edge(1, 4)
>>> g.add_edge(4, 2)
>>> g.add_edge(5, 4)
>>> g.add_edge(5, 5)
```

```
>>> g.destinations(0)
[1, 2]
>>> g.destinations(5)
[4, 5]
>>> g.destinations(3)
[]
>>> g.has_path(0,2)  # 0 -> 2
True
>>> g.has_path(5,1)  # 5 -> 4 -> 2 -> 0 -> 1
True
>>> g.has_path(3,3)  # 3
True
>>> g.has_path(1,5)  # No path exists
False
```

(a) (2.0 points)

Implement the `destinations` method of the `Graph` class.

```
class Graph:
    "A graph."

    def __init__(self, nodes):
        self.nodes = nodes
        self.edges = []

    def add_edge(self, source, destination):
        "Add an edge from source to destination."
        if source >= 0 and source < self.nodes:
            if destination >= 0 and destination < self.nodes:
                if (source, destination) not in self.edges:
                    self.edges.append((source, destination))

    def destinations(self, source):
        "Returns a list of all destinations of the given node."
        assert source >= 0 and source < self.nodes
        return ___(a)___

    def has_path(self, source, destination):
        "Returns True if a path exists from source to destination, and False otherwise."
        ...  # The implementation of this method is omitted, but assume it works.
```

i. (2.0 pt) Fill in blank (a).

```
[i[1] for i in self.edges if i[0] == source]
```

(b) (6.0 points)

The function `compose_path` takes in a list of functions `funcs` and three non-negative integers, `x`, `y`, and `maxval`. It returns whether there exists a list of numbers `s` such that:

- `s` begins with `x` and ends with `y`.
- `s[i+1] = f(s[i])` for some `f` in `funcs`, for every adjacent pair of elements `s[i]` and `s[i+1]`.
- `0 <= s[i]` and `s[i] < maxval` for all elements `s[i]`.

Assume `x` and `y` are less than `maxval` and that all elements of `funcs` are one-argument functions that take in an integer and output an integer. Complete the implementation of `compose_path`. (**Hint**: Treat each function call f(x) as "moving" from x to f(x).)

```
def compose_path(funcs, x, y, maxval):
    """
    >>> f, g, h = lambda x: 2*x-1, lambda x: x*x+1, lambda x: x-4
    >>> #3->10->6->11->122->118->114->110->106->102->98
    >>> #  g   h  f   g    h    h    h    h    h    h
    >>> compose_path([f, g, h], 3, 98, 1000)
    True
    >>> #The above path hits 122, and no other path exists to get from 3 to 98
    >>> compose_path([f, g, h], 3, 98, 122)
    False
    >>> [i for i in range(100) if compose_path([h], 10, i, 100)]
    [2, 6, 10]
    """
    g = Graph(___(a)___)
    for f in funcs:
        ___(b)___:
            ___(c)___
    ___(d)___
```

i. (1.0 pt) Fill in blank (a).

```
maxval
```

ii. (1.0 pt) Fill in blank (b).

```
for j in range(maxval)
```

iii. (2.0 pt) Fill in blank (c).

```
g.add_edge(j, f(j))
```

iv. (1.0 pt) Fill in blank (d).

`return g.has_path(x, y)` **The graph contains all allowed transitions j -> f(j) after iterating through all `f` and all `j`. Therefore, `has_path` returns whether a sequence of allowed transitions exists.**

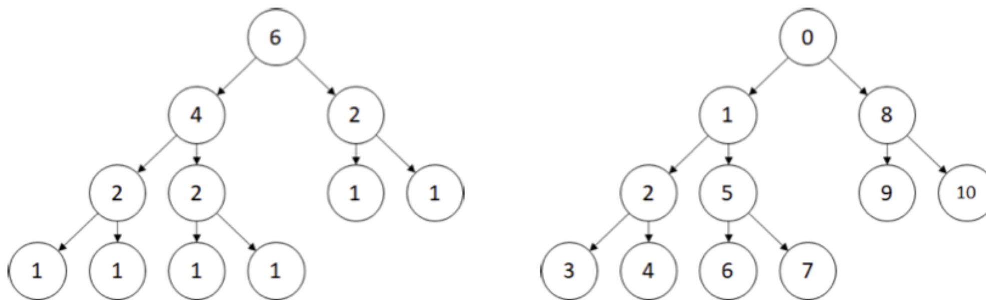
7. (0.0 points) A+ Question 2: Deforestation

This A+ question is not worth any points. It can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first!

Implement the function `tree_to_graph`, which takes as input a Tree instance. It returns a Graph instance (defined in the previous question) corresponding to the tree, with nodes ordered in pre-order starting with 0. Pre-order is defined as:

- The root node is assigned the lowest value unassigned.
- After this, the first branch of the tree is assigned values according to pre-order.
- Then, the second branch of the tree is assigned values, and so on.

For example, calling `tree_to_graph` on this `Tree` instance to the left should create this `Graph` instance to the right.



```
def tree_to_graph(tree):
    def helper(tree, op1, op2):
        a = op1()
        return op2(a, [helper(b, op1, op2) for b in tree.branches]) or a
    num_nodes = helper(tree, ___(a)___, ___(b)___)
    g = Graph(num_nodes)
    b = iter(range(num_nodes))
    helper(tree, ___(c)___,___(d)___)
    return g
```

(a) (0.0 pt) Fill in blank (a).

```
lambda: 1
```

(b) (0.0 pt) Fill in blank (b).

```
lambda a, b: a+sum(b)
```

(c) (0.0 pt) Fill in blank (c).

```
lambda: next(b)
```

(d) (0.0 pt) Fill in blank (d).

```
lambda a, b: any([g.add_edge(a,i) for i in b])
```

See the following page for an explanation.

8. (0.0 points) OPTIONAL

(a) (0.0 pt) Draw a picture related to CS 61A or write a program about CS 61A.

A+ question explanation: There are three parts to this question:

- **Determining the number of nodes in the tree (blank a and b)**
- **Creating a graph with the given number of nodes (blank c)**
- **Adding the appropriate edges to the graph (blank d and e)**

Step 2 can be done fairly easily by calling `Graph(numnodes)`

For Steps 1 and 3, the tricky part is that the requested behavior must be achieved solely through the two lambda function inputs of the helper.

Step 1 can be done by using `op1` as a lambda that returns 1 and `op2` as a function to add the results of all branches

Step 3 is trickier, but can be done if we make `op1` return the ID of the graph node corresponding to the current tree node. As it turns out, the pre-order definition perfectly matches assigning IDs in numeric order (using the traversal order given in the helper function), so we can use the iterator `b` to perform this. `op2` is then set to run `add_edge` for each element of the branches, and return a falsey value so that we output a. Note that the solution provides must use `any` instead of `all`, in order to return False on leaves.

No more questions.