CS 61A Structure and Interpretation of Computer Programs
Spring 2026

MIDTERM

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?

(d) Sign (or type) your name to confirm that all work on this exam will be your own. The penalty for academic misconduct on an exam is an F in the course.

1. (2.0 points) Finding Errors

Sofia is trying to write some code to represent a rectangular prism in Python. This is her code:

```
1  class RectangularPrism:
2      def __init__(self, width, length, height):
3          self.width = width
4          self.length = length
5          self.height = height
6
7      def base_area(self):
8          return self.width * self.length
9
10     def is_square_base(self):
11         return self.width == self.length
12
13     def volume(self):
14         return self.base_area * self.height
```

(a) (1.0 pt) Sofia runs the following code expecting to see it return approximately 400, the volume of a 10 by 10 by 4 rectangular prism.

```
>>> sq = RectangularPrism(10, 10, 4)
>>> sq.volume()
```

Unfortunately, the code results in this error instead:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/sofia/rectangularprism.py", line 14, in volume
    return self.base_area * self.height
TypeError: unsupported operand type(s) for * : 'method' and 'int'
```

According to this traceback, which method contains an error?

◯ `__init__`

◯ `base_area`

◯ `is_square_base`

◯ `volume`

(b) (1.0 pt) Rewrite the buggy line of code from the method that you identified above to fix the error and return the correct value.

2. (5.0 points) Hog Revisited

(a) (2.0 pt) In the Hog project, the `roll_dice(num_rolls, dice)` function simulates rolling `dice` a given number of `num_rolls` times and returning the results. According to the standard rules of Hog, if any of the dice rolled shows a 1, the player's score for that turn is exactly 1, regardless of the other dice. Otherwise, the player's score is the sum of all the dice rolls.

```
def fixed_dice():
    while True:
        yield 3
        yield 4
        yield 1
        yield 5

die = fixed_dice()
result1 = roll_dice(3, die)
result2 = roll_dice(3, die)
result = result1 + result2
```

Assume `roll_dice` is implemented correctly and this code runs without any errors. What does `result` evaluate to?

(b) (1.0 pt) In Hog, a *strategy* is defined as a function that takes two arguments (the current player's score and the opponent's score) and returns the number of dice the current player decides to roll. Consider the following function that returns a strategy:

```
def make_catch_up_strategy(base_rolls):
    """Returns a strategy that rolls more dice if the player is losing."""
    def strategy(score, opponent_score):
        if score < opponent_score:
            return base_rolls + 2
        return base_rolls
    return strategy

my_strat = make_catch_up_strategy(4)
dice_to_roll = my_strat(25, 30)
```

What does `dice_to_roll` evaluate to?

◯ 2

◯ 4

◯ 6

◯ This code errors because `make_catch_up_strategy` takes one argument but we passed two into `my_strat`.

(c) (2.0 pt) In Hog, we implemented `make_averaged`, a higher-order function that takes in an `original_function` and a `times_called`, and returns a new function. We have simplified this returned function such that it repeatedly calls the `original_function` and returns the average of the results.

Note: This is slightly different from what you implemented in the project, with the code being simpler.

```
def make_averaged(original_function, times_called=1000):
    def returned_function():
        total = 0
        for _ in range(times_called):
            total += original_function()
        return total / times_called
    return returned_function

# Assume this test dice alternates between returning 4 and 3
def alternating_dice():
    # Implementation omitted, infinitely returns 4, 3, 4, 3...

averaged_roll = make_averaged(alternating_dice)
final_result = averaged_roll()
```

What does `final_result` evaluate to? If an error occurs, write `ERROR`.

3. (3.0 points) Branching Out

Consider the following list-based construction (the full definition, including implementation of `is_tree`, is on Page 2 left column of Midterm 2 Study Guide):

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_leaf(tree):
    return not branches(tree)
```

Finish implementing the `last_leaf` function to return the label of the last leaf in the given tree. The last leaf is the rightmost leaf: the one reached by always following the last branch in a tree's list of branches until a leaf is found.

```
def last_leaf(t):
    """
    >>> t = tree('a', [tree('b', [tree('d'), tree('e')]), tree('c')])
    >>> t
    ['a', ['b', ['d'], ['e']], ['c']]
    >>> last_leaf(t)
    'c'
    >>> t = tree(5, [
                    tree(2, [tree(7), tree(1, [tree(0), tree(4)]), tree(9)]),
                    tree(3, [tree(6), tree(8)])
            ])
    >>> t
    [5, [2, [7], [1, [0], [4]], [9]], [3, [6], [8]]]
    >>> last_leaf(t)
    8
    """
    if is_leaf(t):
        _________
           (a)
    last_branch = _________
                     (b)
    return last_leaf(last_branch)
```

(a) (1.5 pt) Fill in blank (a).

(b) (1.5 pt) Fill in blank (b).

4. (9.0 points) Teaching Mutating Lists of Hurdles

Assume the following code has been run (you may find drawing box-and-pointer diagrams helpful). Recall the `pop` method of a list removes and returns the last element of the list; if it is provided an argument, `pop(i)` removes and returns the element at index `i`.

```
lst1 = [1, 2, 3, 4, 5]
lst2 = lst1[:3]
lst3 = lst1
lst1.append(lst2)
lst3.extend([lst2.pop() for _ in range(2)])
lst2[0] = 'c'
lst4 = lst1[:]

def mystery(lst):
    for i in range(len(lst)):
        print(lst[i])
        if lst[i] < 0:
            lst.pop(i)
```

For each part below, write the output of running the given code.

(a) (0.5 pt)

```
>>> lst1 is lst3
```

(b) (1.0 pt)

```
>>> lst1 is lst4
```

(c) (1.0 pt)

```
>>> lst1 == lst4
```

(d) (4.0 pt)

```
>>> lst1
```

(e) (2.5 pt) Write what Python would print out from the following run of the `mystery` function.

If an error occurs, you should also include it. The error does not have to match exactly what Python prints out, but it should include the error type. e.g. `SyntaxError`, `ZeroDivisionError`, etc.

```
>>> mystery([-1, -3, 4, -2, 7, 3])
```

5. (9.0 points) Making Pen Friends!

A `Pen` has an attribute for the `animals` (dictionary) it holds. An `Animal` has attributes for its unique `name` (str), the `pen` (`Pen`) it belongs to, and a `list` of its `friends`.

An `Animal` may only `befriend` another `Animal` that belongs to the same `Pen`, in which case they become friends and are now on each other's list of friends.

```
class Pen:
    def __init__(self):
        self.animals = {}

class Animal:
    """An Animal has a befriend method that allows animals to make friends.
    >>> my_pen = Pen()
    >>> other_pen = Pen()
    >>> alice = Chicken("Alice", my_pen)
    >>> bob = Chicken("Bob", other_pen)
    >>> clarice = Chicken("Clarice", my_pen)

    >>> my_pen.animals
    {'Alice': Chicken(Alice), 'Clarice': Chicken(Clarice)}
    >>> other_pen.animals
    {'Bob': Chicken(Bob)}
    >>> clarice.friends
    []
    >>> clarice.befriend(bob)
    Cannot befriend an animal that doesn't share a pen!

    >>> clarice.befriend(alice)
    Clarice made a new friend!
    >>> alice.friends
    ['Clarice']
    """
    sound = "silent"
    def __init__(self, name, pen):
        self.name = name
        self.pen = pen
        _______
          (c)
        self.friends = _______
                         (d)

    def befriend(self, other):
        if _______:
             (e)
            print("Cannot befriend an animal that doesn't share a pen!")
        else:
            _______
              (f)
            _______
              (g)
            print(f"{self.name} made a new friend!")

    def describe(self):
        return f'{self.name} goes {self.sound}'
```

```
    def __repr__(self):
        return f'Animal({self.name})'

class Chicken(Animal):
    sound = 'bawk'

    def __repr__(self):
        return f'Chicken({self.name})'

class BabyChick(Chicken):
    def __init__(self, name, pen, mother):
        super().__init__(name, pen)
        self.mother = mother

    def describe(self):
        return super().describe() + f', hatched by {self.mother}'
```

What would Python output if the following lines of code were run?

(a) (0.5 pt)

```
>>> my_pen = Pen()
>>> bertha = Chicken("Bertha", my_pen)
>>> repr(bertha)
```

(b) (1.5 pt)

```
>>> larry = BabyChick("Larry", my_pen, bertha)
>>> larry.describe()
```

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

(d) (0.5 pt) Fill in blank (d).

(e) (1.5 pt) Fill in blank (e).

(f) (1.5 pt) Fill in blank (f).

(g) (1.5 pt) Fill in blank (g).

6. (6.0 points) When The Last Leaf Falls...

Recall that the built-in `iter` function uses the `__iter__` method of an object to return an iterator for that object. Consider the following tree class's iteration method and the specific Tree instance that is created below (drawing it may be helpful).

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = branches

    def __iter__(self):
        yield self.label

        for branch in self.branches:
            yield from iter(branch)

t = Tree(5, [Tree(2, [Tree(7), Tree(1, [Tree(0), Tree(4)]), Tree(9)]), Tree(3, [Tree(6), Tree(8)])])

it = iter(t)
print(next(it)) # FIRST PRINT STATEMENT
next(it)
next(it)
next(it)
next(it)
print(next(it)) # SECOND PRINT STATEMENT
```

(a) (1.0 pt) What is the output of the first print statement in the above code?

(b) (3.0 pt) What is the output of the second print statement in the above code?

(c) (2.0 pt) How many *more* times can `next(it)` be called after the second print statement *before* the `StopIteration` exception is raised?

Write one of the following:

- an integer
- `StopIteration` if we've already hit the StopIteration exception
- `Infinitely many` if it can be called an infinite number of times

7. (13.0 points) The Power of Friendship

You and your friends want to make friendship earrings of `s` beads. Each friend has uniquely colored beads and everyone sits in a circle taking turns putting beads onto the end of an earring. Each friend has their own speed that they can put beads on. The list `limit` records the top speed of each friend: On friend `i`'s turn, they put on at least 1 bead and up to and including their top speed `limit[i]` amount of beads. Formally,

(a) An earring must have exactly `s` beads.
(b) Each friend `i`, where `0 <= i < len(limit)`, can place `m` beads onto the earring, where `1 <= m <= limit[i]` on their turn.

Implement `circle` and `earring` to return the **total number of ways to create an earring**. Assume all friends have a `limit[i]` of at least 1 and `len(limit) >= 2`. You are friend 0 and put the first bead(s) on.

```
def circle(f, n):
    """Return the next friend given the current friend f and total people in the circle n.
    >>> circle(0, 2)
    1
    >>> circle(4, 5)
    0
    >>> circle(3, 6)
    4
    """
    return _______
             (a)


def earring(s, limit):
    """Return the total number of earrings of length s.

    In the doctests, each dashed sequence represents a potential earring, where each dash separates
    a friend's turn and the number of beads they put on in their turn.

    >>> earring(3, [1, 2]) # 1-2, 1-1-1
    2
    >>> earring(4, [2, 1]) # 1-1-2, 2-1-1, 1-1-1-1
    3
    >>> earring(5, [1, 1, 1, 1, 1]) # 1-1-1-1-1
    1
    """
    def helper(friend, length_left):
        if _______:
             (b)
            return 1

        next_f = _______
                   (c)

        return _______([_______ for m in range(_______, min(_______) + 1)])
                 (d)      (e)                    (f)          (g)

    return _______
             (h)
```

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

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

(c) **(3.0 pt)** Fill in blank (c).

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

- ◯ `len`
- ◯ `sum`
- ◯ `any`
- ◯ `all`
- ◯ `max`
- ◯ `min`

(e) **(3.0 pt)** Fill in blank (e).

(f) **(1.0 pt)** Fill in blank (f).

(g) **(2.0 pt)** Fill in blank (g).

(h) **(1.0 pt)** Fill in blank (h).

8. (5.0 points) Linking A List

Now that Angel has learned the efficiency of removing elements from linked lists, they have decided to convert all of their lists to linked lists. Write the function `convert` that takes in a list and returns a linked list with the elements in the same order. The `Link` class is defined on the Midterm 2 Study Guide (Page 2 left column).

Implement the function below. It should return a linked list with the same elements as the input list as they are: **If an element is a list, it should be put in the linked list *as is* and should *not* be converted to a linked list.** Use the doctests to guide your implementation and recall that `Link.empty = ()`:

```
def convert(lst):
    """
    >>> convert([1, 2, 3])
    Link(1, Link(2, Link(3)))
    >>> convert([1, [2, 3], 4])
    Link(1, Link([2, 3], Link(4)))
    >>> convert([])
    ()
    """
```

(a) (5.0 pt) Complete the `convert` function.

9. (3.0 points) Off To The Races

(a) (1.0 pt) The goal of the `remove_long_words1` function below is to remove all words in the linked list `links` whose length is greater than or equal to `length` (assume that each link's `first` attribute is a string).

```
def remove_long_words1(links, length):
    """
    >>> links = Link("hi", Link("hello", Link("hey", Link("howdy"))))
    >>> remove_long_words1(links, 4)
    Link('hi', Link('hey'))
    """

    if links is Link.empty:
        return Link.empty
    if len(links.first) >= length:
        return remove_long_words1(links.rest, length)
    return Link(links.first, remove_long_words1(links.rest, length))
```

What is the order of growth of `remove_long_words1` with respect to the length of `links` (i.e. the number of words linked together with links)?

◯ Constant

◯ Logarithmic

◯ Linear

◯ Quadratic

◯ Exponential

(b) (1.0 pt) The next function, `remove_long_words2`, serves the same purpose but is implemented slightly differently.

```
def remove_long_words2(links, length):
    """
    >>> links = Link("hi", Link("hello", Link("hey", Link("howdy"))))
    >>> remove_long_words2(links, 4)
    Link('hi', Link('hey'))
    """

    head = links
    while head is not Link.empty and len(head.first) >= length:
        head = head.rest

    if head is Link.empty:
        return Link.empty

    prev = head
    curr = head.rest

    while curr is not Link.empty:
        if len(curr.first) >= length:
            prev.rest = curr.rest
        else:
            prev = curr
        curr = curr.rest

    return head
```

What is the order of growth of the alternative function, `remove_long_words2`, with respect to the length of `links`?‘

◯ Constant

◯ Logarithmic

◯ Linear

◯ Quadratic

◯ Exponential

(c) (1.0 pt) Which of the functions requires more **space** to run?

◯ `remove_long_words1`

◯ `remove_long_words2`

◯ They both require the same amount of space.

No more questions.