CS 61A Fall 2025

Structure and Interpretation of Computer Programs

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?

(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. (5.0 points) What Would Python Display?

Assume the following code has been executed.

```
def weird(s):
    if s:
        if len(s) > 1:
            yield s[1]
        for x in weird(s[1:]):
            yield x
        yield s[0]

t = weird([5, [6, 7], 8])

def even(s):
    def weirder(t, i):
        if i < len(s):
            t.append(s[i-1])
            t.append(s[i])
            weirder(t, i * 2)
        return t
    return weirder([], 1)
```

Write the output displayed by each expression below or *Error* if an error occurs. If some output is displayed before the error, include it. Assume that each expression is evaluated in order and sequentially, so evaluating the first could affect the value of the second. Write *Generator* for a generator object and *Function* for a function.

(a) (1.0 pt) `print(next(t))`

- ○ 5
- ○ 6
- ○ 7
- ● [6, 7]

(b) (3.0 pt) `print([x for x in t])`

```
[8, 8, [6, 7], 5]
```

(c) (1.0 pt) What is the order of growth of the time it takes to execute `even(s)` for a list of numbers `s` in terms of the length of `s`? Assume that `len`, `list.append`, accessing a list item using square brackets, and all arithmetic operators (such as `-`, `<`, and `*`) each takes a constant amount of time.

- ○ exponential
- ○ quadratic
- ○ linear
- ● logarithmic
- ○ constant

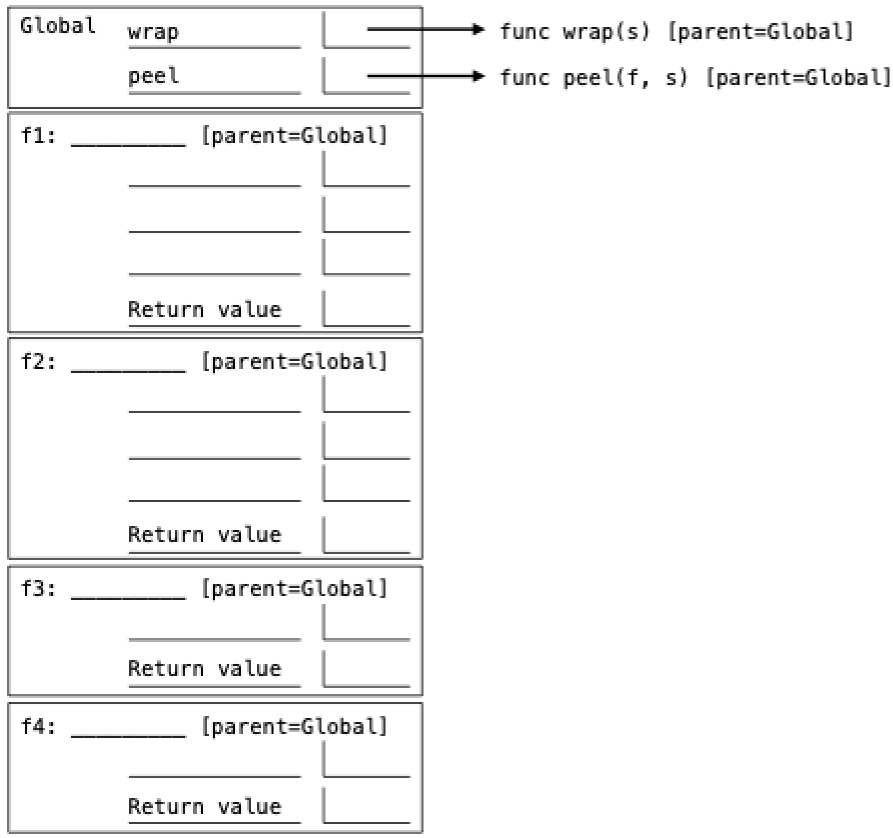
2. (4.0 points) That's a Wrap

Answer the question below about this code. Draw an environment diagram, but the diagram itself will not be scored.

```
def wrap(s):
    return [s[0], s]

def peel(f, s):
    f(f(s)).append(2)
    t = s[1]
    t[0] = 3
    t = [4]
    return s

s = [5, [6, [7]]]
print(peel(lambda s: s[1],
           wrap(s)))
```



(a) (4.0 pt) What is displayed by running this code? If the code errors, write *Error*.

```
[5, [3, [6, [7], 2]]]
```

3. (5.0 points) Inclusive

Definition: A tree `whole` *includes* a tree `part` if it is possible to create a tree with the same `repr` string as `part` by removing some of the nodes of `whole` (but making no other changes). When a node is removed, all of its children are removed as well. The order of the nodes that remain must match, so that the `repr` string is the same.

Implement `includes`, which takes a `Tree` instance `part` and a `Tree` instance `whole` and returns whether `whole` includes `part`. The `Tree` class appears on the Midterm 2 Study Guide (Page 2 left column).

```
def includes(part, whole):
    """Return whether the Tree part is included in the Tree whole.

    >>> p1 = Tree(2, [Tree(20), Tree(22)])
    >>> right = Tree(2, [Tree(19), Tree(20), Tree(21), Tree(22), Tree(23)])
    >>> wrong = Tree(2, [Tree(19), Tree(22), Tree(20), Tree(23)])
    >>> [includes(p1, right), includes(p1, wrong), includes(Tree(1, [right]), p1)]
    [True, False, False]
    >>> p2 = Tree(1, [Tree(6), p1])
    >>> includes(p2, Tree(1, [Tree(5), Tree(6, [Tree(7)]), wrong, right]))
    True
    >>> includes(p2, Tree(1, [Tree(5), Tree(6, [Tree(7)]), wrong, Tree(8, [right])]))
    False
    """
    if whole.label != part.label:
        return False
    p, w = 0, 0
    while _______:
            (a)

        if _______:
             (b)

            p += 1
        w += 1
    return _______
             (c)
```

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

- ○ `p < len(part.branches)`
- ○ `w < len(whole.branches)`
- ○ `p < len(part.branches) or w < len(whole.branches)`
- ● `p < len(part.branches) and w < len(whole.branches)`

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

```
includes(part.branches[p], whole.branches[w])
```

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

```
p == len(part.branches)
```

4. (6.0 points) Exclusive

(a) (2.0 pt) Implement `exclude`, which takes a list of numbers `s` and a number `x`. It returns a list with all the elements of `s` that are not equal to `x`. The input list should not be modified. It is possible that `s` could be an empty list.

Write your answer directly in the blank below.

```
def exclude(s, x):
    """Return a list with all of the elements of s except those equal to x.

    >>> a = [3, 4, 5, 3.0, 6, 5, 3]
    >>> exclude(a, 3)
    [4, 5, 6, 5]
    >>> a  # no change to a
    [3, 4, 5, 3.0, 6, 5, 3]
    """

    return ________________________________________________________________________________
```

(b) (4.0 pt) Implement `exclude_link`, which takes a linked list of numbers `s` and a number `x`. It returns a linked list with all the elements of `s` that are not equal to `x`. The input linked list should not be modified. It is possible that `s` could be an empty linked list. The `Link` class appears on the Midterm 2 Study Guide (Page 2 left column).

Write your answers directly in the blanks below.

```
def exclude_link(s, x):
    """Return a linked list with all of the elements of linked list s except those equal to x.

    >>> a = Link(3, Link(4, Link(5, Link(3.0, Link(6, Link(5, Link(3)))))))
    >>> exclude_link(a, 3)
    Link(4, Link(5, Link(6, Link(5))))
    >>> a  # no change to a
    Link(3, Link(4, Link(5, Link(3.0, Link(6, Link(5, Link(3)))))))
    """

    if ____________________________________________________________________________________:

        return ____________________________________________________________________________

    elif __________________________________________________________________________________:

        return ____________________________________________________________________________

    else:

        return ____________________________________________________________________________
```

(a) Implement `exclude`.

```
[y for y in s if y != x]
```

(b) Fill in the first blank of `exclude_link` (if condition).

```
s is Link.empty
```

(c) Fill in the second blank of `exclude_link` (first return statement).

```
s
```

(d) Fill in the third blank of `exclude_link` (elif condition).

```
s.first == x
```

(e) Fill in the fourth blank of `exclude_link` (elif return statement).

```
exclude_link(s.rest, x)
```

(f) Fill in the fifth blank of `exclude_link` (else return statement).

```
Link(s.first, exclude_link(s.rest, x))
```

5. (10.0 points) Pizza Parlor

A Pizza `Parlor` has attributes for its `name` (`str`), the `price` (`int`) of each pizza it sells, the number of `pizzas` (`int`) it has available to cook, an `alt` (an alternative `Parlor`) to help with large orders, and its total `profit` (`int`). Its `order` method takes a positive integer `quantity` and optionally a `Parlor` instance `source` and processes an order as follows. A `Parlor` will `cook` at most its number of `pizzas`. If an order is larger than its number of `pizzas`, it will `order` the rest from its `alt`. Its `profit` increases by its `price` times the `quantity` of pizzas ordered, but decreases by its `alt`'s `price` times the quantity it orders from `alt`. A message is printed each time an order is completed or made with another `Parlor` as the `source`. A `MegaParlor` can `cook` any quantity of pizzas and has no `alt`.

```
class Parlor:
    """A Pizza Parlor has an alt(ernative) Parlor that helps it handle big orders.

    >>> c = Parlor('Sliver', 3, 10, Parlor('Cheeseboard', 4, 10, MegaParlor(5)))
    >>> c.order(6)  # All $3*6 goes to Sliver; it has 4 pizzas left
    Sliver now has $18 profit
    >>> c.order(6)  # Another $3*6 goes to Sliver, which pays $4*2 to Cheeseboard
    Sliver forwarded an order of 2 to Cheeseboard
    Cheeseboard now has $8 profit
    Sliver now has $28 profit
    >>> c.order(10) # Sliver gets $3*10 but pays Cheeseboard $4*10, which pays $5*2 to MEGA
    Sliver forwarded an order of 10 to Cheeseboard
    Cheeseboard forwarded an order of 2 to MEGA
    MEGA now has $10 profit
    Cheeseboard now has $38 profit
    Sliver now has $18 profit
    """
    def __init__(self, name, price, pizzas, alt):
        self.name, self.price, self.pizzas, self.alt = name, price, pizzas, alt
        self.profit = 0

    def cook(self, q):
        overload = _______                # The quantity that must be ordered from alt
                      (a)
        self.pizzas -= (q - overload)   # The quantity cooked by this Parlor
        return overload

    def order(self, quantity, source=None):
        if isinstance(source, Parlor):
            print(source.name, 'forwarded an order of', quantity, 'to', self.name)
        self.profit += _______
        rest = _______   (b)
                  (c)
        if rest:
            self.profit -= _______
            _______          (d)
              (e)
        print(f'{self.name} now has ${self.profit} profit')

class MegaParlor(Parlor):
    def __init__(self, price):
        self.name, self.price, self.profit = "MEGA", price, 0

    def cook(self, q):
        return _______
```

```
(f)
```

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

- ○ `q - self.pizzas`
- ○ `abs(q - self.pizzas)`
- ● `max(0, q - self.pizzas)`
- ○ `min(0, q - self.pizzas)`

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

- ○ `price * quantity`
- ○ `price * self.quantity`
- ○ `price * pizzas`
- ○ `price * self.pizzas`
- ● `self.price * quantity`
- ○ `self.price * self.quantity`
- ○ `self.price * pizzas`
- ○ `self.price * self.pizzas`

(c) **(2.0 pt)** Fill in blank (c). **IMPORTANT:** You **may not** call `type` or `isinstance`.

```
self.cook(quantity)
```

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

```
rest * self.alt.price
```

(e) **(3.0 pt)** Fill in blank (e). **IMPORTANT:** You **may not** write `=`.

```
self.alt.order(rest, self)
```

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

```
0
```

6. (20.0 points) Two Topping Pizzas

Kay and John want to share a pizza. Kay likes mushroom and John likes pineapple, but they won't eat a pizza that has a mushroom slice next to a pineapple slice (eek!). A pizza with `n` slices is represented as a length-`n` string containing `M` for a mushroom slice, `P` for a pineapple slice, and `_` for a slice with no topping. Slices are *next to* each other if they are adjacent in the string **or** are the first and last element of the string. Kay and John won't eat `MMMMPPPP` (because `MP`) or `MM__PP` (because the first `M` is next to the last `P`), but will eat these: `'MM__M_PP_M'`, `'_P_M_P_M'`, and `'_PP_MM_P_M_'`.

(a) (2.0 points)

Implement `symmetrical`, which takes a dictionary `d` and returns `True` if for every pair `(k, v)` for which `d[k] == v`, it's also true that `d[v] == k`, and returns `False` otherwise. Assume `None` is not a key or value in `d`.

The `get` method of a `dict` takes a key and returns the value for that key if the key is in the dictionary and otherwise returns `None`.

```
def symmetrical(d):
    """Return whether every key-value pair in d is also a value-key pair.

    >>> symmetrical({'M': 'P', 'P': 'M', 'G': 'G'})
    True
    >>> symmetrical({'M': 'P', 'P': 'M', 'G': 'M'})  # No M->G
    False
    >>> symmetrical({'M': 'P', 'P': 'M', 'G': 'T'})  # No T->G
    False
    """
    return all([ _______ for k in d.keys()])
                   (a)
```

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

- ○ `k in d and d[k] in d`
- ○ `k in d.keys() and d[k] in d.values()`
- ○ `k in d.keys() and v in d.values()`
- ○ `d[k] == d.get(k)`
- ○ `d[k] == d.get(v)`
- ○ `d[k] == v and d.get(v) == k`
- ○ `d[k] != v or d.get(v) == k`
- ○ `d[k].get(v) == v`
- ○ `d[k].get(v) == k`
- ○ `d[k].get(k) == v`
- ○ `d[k].get(k) == k`
- ○ `d.get(d[k]) == v`
- ● `d.get(d[k]) == k`
- ○ `d.get(d[k]) == d.get(v)`
- ○ `d.get(d[k]) == d.get(k)`

(b) (4.0 points)

Implement `acceptable`, which takes a non-empty string `pizza` and a symmetrical dictionary of strings `disallow`. It returns whether `pizza` represents a pizza in which no slice that is next to another slice appears as a key-value pair in `disallow`. Assume that every key in `disallow` is not equal to its value.

```
def acceptable(pizza, disallow={'M': 'P', 'P': 'M'}):
    """Return whether there are no slices next to each other with a disallowed topping pair.

    >>> acceptable("MM_PPP__M")
    True
    >>> acceptable("MM__PPP")  # The first slice M and last slice P are next to each other.
    False
    >>> acceptable("MM__PPP_")
    True
    >>> acceptable("MM__MPPP_")
    False
    """
    assert symmetrical(disallow)
    previous = ''
    for slice in pizza:
        if previous _______:
                       (b)
            return False

        _______
          (c)
    return _______
             (d)
```

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

- ○ `or disallow.get(previous) == slice`
- ● `and disallow.get(previous) == slice`
- ○ `or disallow.get(previous) == disallow.get(slice)`
- ○ `and disallow.get(previous) == disallow.get(slice)`

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

- ● `previous = slice`
- ○ `previous = pizza[slice]`
- ○ `previous = previous + 1`
- ○ `slice = previous`
- ○ `slice = pizza[slice]`
- ○ `slice = disallow.get(previous)`

iii. (2.0 pt) Fill in blank (d). **IMPORTANT:** You **may not** call `acceptable`.

```
disallow.get(slice) != pizza[0]
```

(c) (6.0 points)

Implement `count_pizzas`, which takes a positive integer `num_slices` and returns the number of different pizzas that have mushroom (`M`), pineapple (`P`) or no topping (`_`) slices without `M` next to `P`. You **may not** call `acceptable`.

```
def count_pizzas(num_slices):
    """Return the number of ways to top a pizza with num_slices.
    >>> count_pizzas(1) # One slice can have mushroom, or pineapple, or no topping.
    3
    >>> count_pizzas(2) # MM  M_  _M  __  PP  P_  _P
    7
    """
    def f(n, first, previous):
        if _______:
             (e)
            if first == previous or _______:
                return 1               (f)
            return 0
        allowed_next = _______
                          (g)
        if previous == 'P':
            allowed_next.remove('M')
        if previous == 'M':
            allowed_next.remove('P')
        return sum([_______ for x in allowed_next])
                       (h)
    slice_types = ['_', 'M', 'P']
    return sum([f(1, x, x) for x in slice_types])
```

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

`n == num-slices`

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

- ● `'_' in [first, previous]`
- ○ `first in ['M', 'P']`
- ○ `first in ['M', 'P'] or previous in ['M', 'P']`
- ○ `first != previous`

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

- ○ `slice_types`
- ● `list(slice_types)`
- ○ `[first, previous]`
- ○ `[first, previous, '_']`

iv. (3.0 pt) Fill in blank (h).

```
f(n+1, first, x)
```

(d) (8.0 points)

Implement `pizzas`, which takes a positive integer `n` and returns a list of all different pizzas that have mushroom (`M`), pineapple (`P`) or no topping (`_`) slices and do not have `M` next to `P`. The `check_ends` argument indicates whether to check that the first and last slice are allowed to be next to each other. You **may not** call `acceptable`.

```
def pizzas(n, check_ends=True):
    """Ways to distribute M and P toppings on a pizza with n slices.

    >>> pizzas(1)
    ['_', 'M', 'P']
    >>> pizzas(2)
    ['__', '_M', '_P', 'M_', 'MM', 'P_', 'PP']
    >>> pizzas(3)
    ['___', '__M', '__P', '_M_', '_MM', '_P_', '_PP',
     'M__', 'M_M', 'MM_', 'MMM', 'P__', 'P_P', 'PP_', 'PPP']
    >>> '_M_P' in pizzas(4)
    True
    >>> [pizzas(8)[i*90] for i in range(1, 7)]  # A few examples from pizzas(8)
    ['___PP_M_', '__P__MMM', '_M__P__P', '_MM_M_P_', '_P_M___M', '_PP_P_MM']
    """
    if n == 0:

        return _______
                 (i)

    recurse = pizzas(n-1, False)

    ways = _______
             (j)

    for y, z in [['M', 'P'], ['P', 'M']]:

        _______([y + x for x in recurse if not x or _______])
          (k)                                          (l)

    ok = lambda x: True

    if check_ends:

        ok = lambda x: x[0] + x[-1] not in ['MP', 'PM']

    return _______  # Must be a call expression
             (m)
```

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

```
[""]
```

ii. (2.0 pt) Fill in blank (j). **IMPORTANT:** You **may not** call `pizzas` in this blank.

```
["-" + x for x in recurse]
```

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

- ○ `yield`
- ○ `yield from`
- ○ `return`
- ○ `ways =`
- ○ `ways.append`
- ● `ways.extend`

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

```
x[0] != z
```

v. (2.0 pt) Fill in blank (m). **IMPORTANT:** *You may only write names, parentheses, and commas.* No [ ].

```
list(filter(ok, ways))
```

vi. (0.0 pt) 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!

Here are two examples of calling `sum` on a list of lists with an optional `start` argument.

```
>>> sum([[1, 2], [3]], start=[])
[1, 2, 3]
>>> sum([], start=[])
[]
```

Fill in the blank of this alternate implementation of `pizzas`. You may call `acceptable` and `count_pizzas`. If your answer is too long, you can write it on multiple lines.

```
def pizzas(n):
    """Ways to distribute M and P toppings on a pizza with n slices."""
    def g(pizzas):
        yield from pizzas
        yield from g(sum([[p + s for s in slices] for p in pizzas], start=[]))
    slices  = ['_', 'M', 'P']
    t, u = _______
    return [next(t) for _ in u]
```

```
filter(lambda p: len(p) == n and acceptable(p), g(slices)),
range(count-pizzas(n))
```

No more questions.