# CS 61A Structure and Interpretation of Computer Programs

# Summer 2024

Midterm 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 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. (12.0 points) Generiterator**

Assume the following code has been executed. No error occurs when executing this code block.

```
def f(x):
    yield from map(lambda x: x[::-1], x)

def next_next(i):
    print(next(i))
    return next(i)

def generiterator1(s):
    yield f(s)
    yield from f(s)
    print('warmed up!')

def generiterator2(s):
    while s:
        yield f(s)
        yield iter(s[0])
        s = s[1:]
    print('generiterating complete!')

g_str = generiterator1(['i', '<3', '61a', '!'])
g_num = generiterator2([[1], [1, 2], [1, 2, 3]])
```

`What Would Python Display`? Write the output displayed by evaluating each expression below. If an error occurs, write `"Error"`, but include all output displayed before the error. If evaluation would run forever, write `"Forever"`. To display an iterator object, write `"Iterator"`. To display a generator object, write `"Generator"`.

Assume the expressions are evaluated in order in the same interactive session, and so evaluating an earlier expression may affect the result of a later one.

**Hint 1:** Draw it out!

**Hint 2:** When a string is passed into `print`, no quotation marks are displayed. When a string is the value of an expression evaluated by the interpreter, quotation marks *are* displayed. A list of strings always displays quotation marks.

**(a) (5.0 points) Jester**

**i. (1.0 pt)** `next(g_str)`

Generator

**ii. (1.0 pt)** `next(g_str)`

```
'i'
```

**iii. (1.0 pt)** `next_next(g_str)`

```
3< 'a16'
```

**iv. (2.0 pt)** `next_next(g_str)`

```
!
warmed up!
```
**Error**

**(b) (7.0 points) Genome**

**i. (1.0 pt)** `list(next(g_num))`

```
[[1], [2, 1], [3, 2, 1]]
```

**ii. (2.0 pt)** `next_next(next(g_num))`

```
1
```
**Error**

**iii. (2.0 pt)** `next_next(next(g_num))`

```
[2, 1]
[3, 2, 1]
```

**iv. (2.0 pt)** `len(list(g_num))`

**generiterating complete!**
```
3
```

**2. (8.0 points) Conveyor Belt**

Draw the environment diagram for the code below using box and pointer notation and then answer the questions that follow. Your diagram will not be graded.

**Hint:** If you pass an argument into `pop`, it uses the argument as the index of the element to remove. If you do not pass an argument into `pop`, it defaults to popping the last element.

```
def conveyor_belt(s):
    k = -1
    while s[k]:
        k = s[0].pop(chain_pop(s[0]))
        s = s[k:]
        if s[0] == box:
            s.append(box or s.append(box))
        else:
            s.append(s.extend(s[:1]))
    return s

def chain_pop(s):
    return s.pop(s.pop(s.pop()))

box = [1, 1, 0, 0]

result = conveyor_belt([box, box[:]])
```

Blank Space for Diagram:

**(a) (4.0 pt)** What would be displayed by evaluating `print(result)` in the Global frame?

```
[[], None, [], None]
```

**(b) (0.5 pt)** What is the result of evaluating `box in result` in the Global frame?

- ● True
- ○ False

**(c) (0.5 pt)** What is the result of evaluating `[] in result` in the Global frame?

- ● True
- ○ False

**(d) (0.5 pt)** What is the result of evaluating `None in result` in the Global frame?

- ● True
- ○ False

**(e) (0.5 pt)** What is the result of evaluating `result[0] == result[1]` in the Global frame?

- ○ True
- ● False
- ○ Error

**(f) (0.5 pt)** What is the result of evaluating `result[0] is result[1]` in the Global frame?

- ○ True
- ● False
- ○ Error

**(g) (0.5 pt)** What is the result of evaluating `result[0] == result[2]` in the Global frame?

- ● True
- ○ False
- ○ Error

**(h) (0.5 pt)** What is the result of evaluating `result[0] is result[2]` in the Global frame?

- ○ True
- ● False
- ○ Error

**(i) (0.5 pt)** What is the result of evaluating `not result[0]` in the Global frame?

- ● True
- ○ False
- ○ Error

**3. (4.0 points) Tree Sum**

Laryn, Charlotte, and Raymond are working together on a CS 61A problem, but they can't agree on a solution. Help them determine which implementation(s) are correct, if any.

`tree_sum` is a function that takes in a tree `t` and a one-argument function `cond` and returns the sum of all the labels, `x`, in `t` for which `cond(x)` returns `True`.

```
>>> t2 = tree(5, [tree(6), tree(7)])
>>> t1 = tree(3, [tree(4), t2])
>>> tree_sum(t1, lambda x: x >= 4)
22
>>> tree_sum(t1, lambda x: x >= 6)
13
>>> tree_sum(t2, lambda x: x >= 6)
13
```

```
# Attempt 1
def tree_sum1(t, cond):
    if cond(label(t)):
        total = label(t)
    else:
        total = 0
    for b in branches(t):
        if cond(label(b)):
            total += tree_sum1(b, cond)
    return total

#########################################

# Attempt 2
def tree_sum2(t, cond):
    if cond(label(t)):
        total = label(t)
    else:
        total = 0
    for b in branches(t):
        total += tree_sum2(b, cond)
    return total

#########################################

# Attempt 3
def tree_sum3(t, cond):
    total = label(t)
    for b in branches(t):
        if cond(label(b)):
            total += tree_sum3(b, cond)
    return total
```

**(a) (4.0 pt)** Which implementation(s) are correct, if any? **Select all** that apply.

- ☐ `tree_sum1` is correct
- ■ `tree_sum2` is correct
- ☐ `tree_sum3` is correct
- ☐ `tree_sum1`, `tree_sum2`, and `tree_sum3` are all incorrect

**4. (17.0 points) Add Consecutive**

Implement `add_consecutive`, a function that takes in a positive integer `n` and returns a list of integers. Each element of the returned list is the sum of adjacent consecutive digits in `n`. Two digits are adjacent if they are directly beside each other. Two digits are consecutive if the absolute difference between them is exactly 1. Two of the same digit are **not** considered consecutive.

**You may not use `str` or `repr` or `[` or `]` or `for`.**

**(a) (9.0 points) Iterative Add Consecutive**

Implement `add_consecutive` iteratively.

```
def add_consecutive(n):
    """
    >>> add_consecutive(123456789)
    [45]
    >>> add_consecutive(567231) # [5 + 6 + 7, 2 + 3, 1]
    [18, 5, 1]
    >>> add_consecutive(111) # repeated digits are not consecutive
    [1, 1, 1]
    >>> add_consecutive(1235689)
    [6, 11, 17]
    >>> add_consecutive(3216598)
    [6, 11, 17]
    >>> add_consecutive(13579)
    [1, 3, 5, 7, 9]
    >>> add_consecutive(12321) # [1 + 2 + 3 + 2 + 1]
    [9]
    >>> add_consecutive(4)
    [4]
    >>> add_consecutive(105)
    [1, 5]
    >>> add_consecutive(135797531)
    [1, 3, 5, 7, 9, 7, 5, 3, 1]
    """
    result = []

    subtotal = 0

    while __________:
             (a)

        rest, last = n // 10, n % 10

        subtotal = __________
                       (b)
        if __________:
               (c)
            result = __________
                         (d)
            subtotal = __________
                           (e)
        n = rest

    result = __________
                 (f)
    return result
```

**i. (2.0 pt) Select all** of the expressions below that could fill in blank (a).

- [ ] `n`
- [ ] `n > 0`
- [ ] `n >= 0`
- [ ] `n != 0`
- [ ] `n > 10`
- [x] `n >= 10`
- [x] `n > 9`
- [ ] `n >= 9`
- [x] `n // 10`
- [ ] `n % 10`

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

```
subtotal + last
```

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

```
abs((rest % 10) - last) != 1
```

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

```
[subtotal] + result
```

**v. (1.0 pt) Select all** of the expressions below that could fill in blank (e).

- [x] `0`
- [ ] `rest`
- [ ] `last`
- [ ] `n`
- [ ] `subtotal`
- [ ] `subtotal + rest`
- [ ] `subtotal + last`
- [ ] `subtotal + n`

**vi. (2.0 pt)** Fill in blank (f).

```
[subtotal + n] + result
```

**(b) (8.0 points) Recursive Add Consecutive**

Implement `add_consecutive` recursively.

```
def add_consecutive(n):
    """
    >>> add_consecutive(123456789)
    [45]
    >>> add_consecutive(567231) # [5 + 6 + 7, 2 + 3, 1]
    [18, 5, 1]
    >>> add_consecutive(111) # repeated digits are not consecutive
    [1, 1, 1]
    >>> add_consecutive(1235689)
    [6, 11, 17]
    >>> add_consecutive(3216598)
    [6, 11, 17]
    >>> add_consecutive(13579)
    [1, 3, 5, 7, 9]
    >>> add_consecutive(12321) # [1 + 2 + 3 + 2 + 1]
    [9]
    >>> add_consecutive(4)
    [4]
    >>> add_consecutive(105)
    [1, 5]
    >>> add_consecutive(135797531)
    [1, 3, 5, 7, 9, 7, 5, 3, 1]
    """
    def helper(n, subtotal):
        rest, last = n // 10, n % 10

        subtotal = __________
                      (a)
        if __________:
              (b)
            return __________
                      (c)
        if __________:
              (d)
            return helper(__________)
                              (e)
        else:
            return __________
                      (f)
    return helper(n, __________)
                        (g)
```

**i. (1.0 pt) Select all** of the expressions below that could fill in blank (a).

- ☐ `0`
- ☐ `rest`
- ☐ `last`
- ☐ `n`
- ☐ `subtotal`
- ☐ `subtotal + rest`
- ■ `subtotal + last`
- ☐ `subtotal + n`

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

```
n < 10
```

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

```
[subtotal]
```

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

```
abs((rest % 10) - last) == 1
```

**v. (1.0 pt)** Fill in blank (e).

```
rest, subtotal
```

**vi. (2.0 pt)** Fill in blank (f).

```
helper(rest, 0) + [subtotal]
```

**vii. (1.0 pt)** Fill in blank (g).

```
0
```

**5. (7.0 points) Combine Tree**

Implement `combine_tree`, a function that takes in a tree `t` and a two-argument function `f` and returns a new tree where the label of any node `b` is the result of calling `f` on the labels of all the nodes in the subtree rooted at `b` (including the label of `b` itself).

You may assume that `f` is an associative function. That is, `f(x, y) == f(y, x)` for all `x` and `y`.

```
def combine_tree(t, f):
    """
    >>> from operator import add, mul
    >>> t = tree(1, [tree(2), tree(3, [tree(4), tree(5), tree(6)])])
    >>> sum_tree = combine_tree(t, add)
    >>> print_tree(sum_tree)
    21
        2
        18
            4
            5
            6
    >>> product_tree = combine_tree(t, mul)
    >>> print_tree(product_tree)
    720
        2
        360
            4
            5
            6
    >>> max_tree = combine_tree(t, max)
    >>> print_tree(max_tree)
    6
        2
        6
            4
            5
            6
    """
    if is_leaf(t):

        return __________
                    (a)
    total = __________
                (b)
    new_branches = []

    for b in branches(t):

        new_b = __________
                     (c)
        new_branches.append(new_b)

        total = __________
                     (d)
    return tree(total, new_branches)
```

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

- (●) `t`
- ( ) `label(t)`
- ( ) `0`
- ( ) `1`

**(b) (1.0 pt) Select all** of the expressions below that could fill in blank (b).

- [ ] `t`
- [x] `label(t)`
- [ ] `0`
- [ ] `1`
- [ ] `f(label(t))`
- [ ] `f(label(t), label(t))`
- [ ] `f(0, label(t))`
- [ ] `f(1, label(t))`
- [ ] `combine_tree(t, f)`
- [ ] `f(sum([combine_tree(b, f) for b in branches(t)]), label(t))`
- [ ] `sum(map(f, [combine_tree(b, f) for b in branches(t)]))`

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

- ( ) `t`
- ( ) `b`
- ( ) `f(t)`
- ( ) `f(b)`
- ( ) `f(label(t))`
- ( ) `f(label(b))`
- ( ) `f(label(t), label(b))`
- ( ) `f(label(b), label(t))`
- ( ) `combine_tree(t, f)`
- (●) `combine_tree(b, f)`

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

```
f(total, label(new_b))
```

**6. (11.0 points) Multi Compose**

Implement `multi_compose`, a function that takes in a list of functions and two integers `x` and `y`. It returns a composite function that on input `x` outputs `y` by applying a subsequence of functions in the input list. Functions can only be applied in the order in which they appear in the list. That is, the ith-indexed function must be applied before the i+1th-indexed function. Each function can be used 0 or 1 times.

If no such composite function exists, return `None`. If more than one such composite function exists, return any one of them.

You may assume `x != y`. Your solution must use `safe_compose`, which composes two functions together when called with two functions as arguments and returns `None` when called with `None` as an argument.

```
def safe_compose(f, g):
    """
    >>> composed = safe_compose(lambda x: x + 1, lambda x: x * 2)
    >>> composed(5)
    11
    >>> safe_compose(lambda x: x, None) is None and safe_compose(None, lambda x: x) is None
    True
    """
    if f is None or g is None:
        return None
    def composed(x):
        return f(g(x))
    return composed


def multi_compose(funcs, x, y):
    """
    >>> add_one, double = lambda x: x + 1, lambda x: x * 2
    >>> sub_three, square = lambda x: x - 3, lambda x: x ** 2
    >>> list_of_funcs = [add_one, double, sub_three, square]
    >>> double_then_square = multi_compose(list_of_funcs, 3, 36)
    >>> double_then_square
    Function
    >>> double_then_square(1) # (1 * 2) ** 2 --> 4
    4
    >>> square_then_double = multi_compose(list_of_funcs, 3, 18)
    >>> square_then_double # None
    >>> all_funcs = multi_compose(list_of_funcs, 3, 25)
    >>> all_funcs(1) # (((1 + 1) * 2) - 3) ** 2 --> 1
    1
    >>> double = multi_compose(list_of_funcs, 50, 100)
    >>> double(1) # 1 * 2 --> 2
    2
    >>> sub_two = multi_compose(list_of_funcs, 50, 48)
    >>> sub_two(2) # 2 - 2 --> 0
    0
    >>> double_then_sub = multi_compose(list_of_funcs, 50, 97)
    >>> double_then_sub(1) # (1 * 2) - 3 --> -1
    -1
    >>> negate = multi_compose(list_of_funcs, 100, -100)
    >>> negate # None
    """
```

```
if x == y:
    return lambda x: __________
                         (a)
if __________:
      (b)
    return None

return __________ or multi_compose(__________)
          (c)                          (d)
```

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

```
x
```

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

```
not funcs
```

**(c)** **(6.0 pt)** Fill in blank (c).

```
safe_compose(multi_compose(funcs[1:], funcs[0](x), y), funcs[0])
```

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

```
funcs[1:], x, y
```

**7. (5.0 points) Memoized Fibonacci Tree**

Recall the Fibonacci sequence from lecture. It is defined as follows:

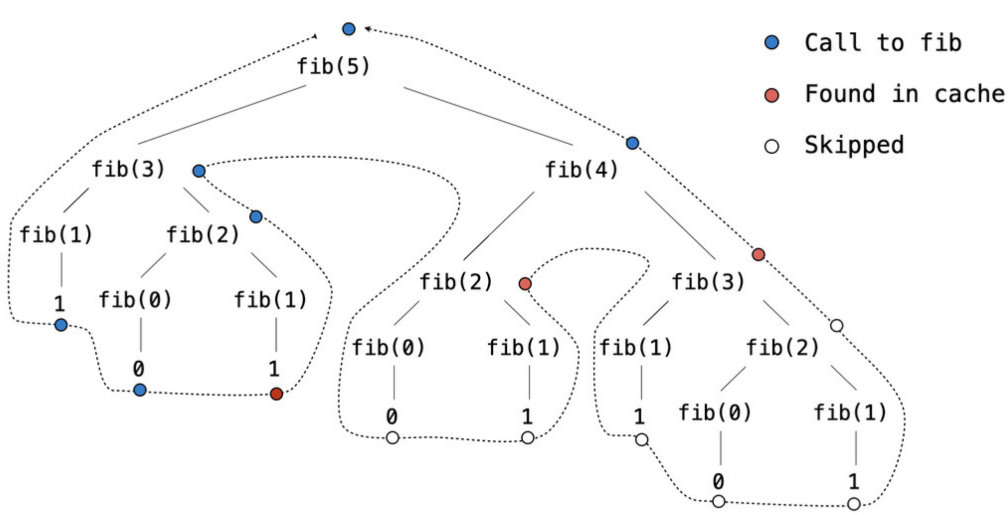
```
         0 if n == 0
fib(n) = 1 if n == 1
         fib(n - 1) + fib(n - 2) else
```

A **Fibonacci Tree** is a tree where each label is a Fibonacci number and each non-leaf node has exactly two children: the two Fibonacci numbers that appear directly before it in the Fibonacci sequence.

The `fib_tree` function below takes in a nonnegative integer `n` and returns a Fibonacci Tree that has root label `fib(n)`. Implement `fib_tree_memo`, a memoized version of `fib_tree`.

```
def fib_tree(n):
    """
    >>> print_tree(fib_tree(5))
    5
        2
            1
            1
                0
                1
        3
            1
                0
                1
            2
                1
                1
                    0
                    1
    """
    # IMPLEMENTATION OMITTED
```

Here is a visual indicating how the call diagram of `fib_tree` should change once you memoize it in `fib_tree_memo`.



```
def fib_tree_memo(n):
    """
    >>> print_tree(fib_tree_memo(5))
    5
        2
            1
            1
                0
                1
        3
            1
            2
    """
    def helper(n, cache):
        if __________:
              (a)
            return tree(__________)
                            (b)
        if n <= 1:

            return tree(n)

        b0, b1 = helper(n - 2, cache), helper(n - 1, cache)

        fib_num = __________
                      (d)
        __________ = fib_num
            (e)
        return tree(fib_num, [b0, b1])

    return helper(n, {})
```

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

```
n in cache
```

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

```
cache[n]
```

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

```
label(b0) + label(b1)
```

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

```
cache[n]
```

**(e) (1.0 pt)** What is the order of growth of the run time of `fib_tree_memo` with respect to `n`?

- ○ Constant, $\Theta(1)$, $O(1)$
- ○ Logarithmic, $\Theta(\log n)$, $O(\log n)$
- ● Linear, $\Theta(n)$, $O(n)$
- ○ Quadratic, $\Theta(n^2)$, $O(n^2)$
- ○ Exponential, $\Theta(b^n)$, $O(b^n)$

**8. (0.0 points) Just for Fun**

This is not for points and will not be graded.

**(a) Optional**: Draw your favorite spot on campus!

**No more questions.**