CS 61A
Fall 2023

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. **(4.0 points)   What Would Python Display?**

   Assume the following code has been executed.

   ```
   s = range(3, 7)
   t = iter(s)
   u = map(lambda x: 2 * x, t)
   v = [next(t), next(t)]      # This line does not cause an error
   ```

   Choose the output **displayed by the interactive Python interpreter** when each expression below is evaluated or *Error* if an error occurs. These expressions are evaluated **in order** and the value of later expressions may be affected by evaluating previous expressions.

   (a) **(1.0 pt)** `[-k for k in v if k in s]`

   ○ `[]`

   ○ `[3, 4]`

   ● `[-3, -4]`

   ○ *Error*

   v contains 3 and 4 (the first two elements of the range 3 through 7), and both are in the range.

   (b) **(1.0 pt)** `tuple(u)`

   ○ `(3, 4, 5, 6)`

   ○ `(3, 4, 5, 6, 7)`

   ○ `(6, 8, 10, 12)`

   ○ `(6, 8, 10, 12, 14)`

   ● `(10, 12)`

   ○ `(10, 12, 14)`

   ○ *Error*

   The map object does not apply its function until `tuple` is called. Making v already iterated over 3 and 4. Thus, the map will begin at 5. The iterator `t` stops after 6.

   (c) **(1.0 pt)** `next(t)`

   ○ `3`

   ○ `5`

   ○ `range(3, 7)`

   ○ `range(5, 7)`

   ● *Error*

   The `tuple` call used up the values in `t`. `StopIteration` occurs.
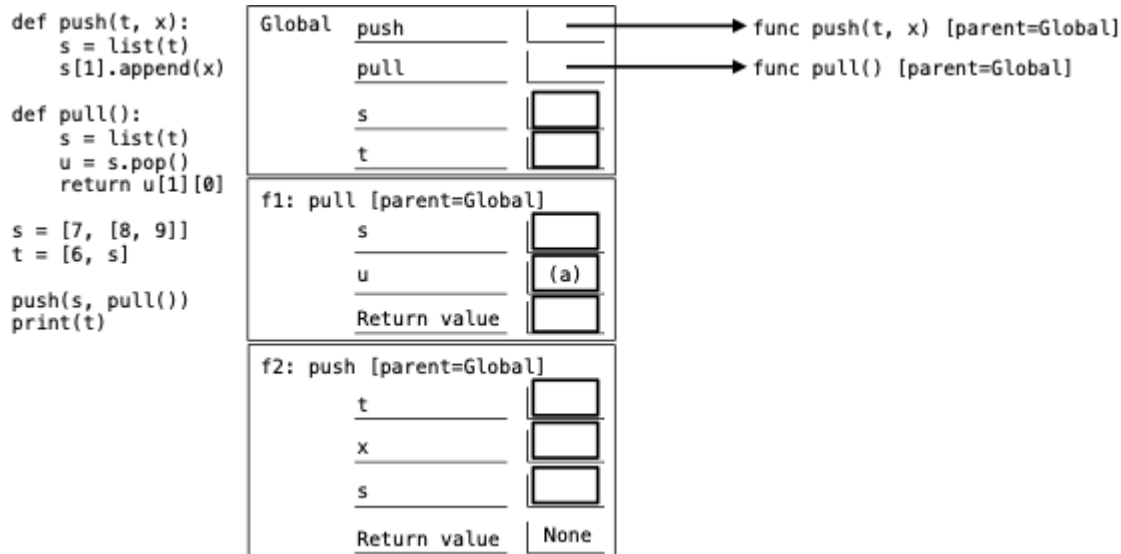
(d) **(1.0 pt)** `next(iter(s)) + next(iter(s))`

- 🔵 6
- ◯ 7
- ◯ 10
- ◯ 11
- ◯ *Error*

One makes a new iterator out of the iterable, range(3,7) in each call, and thus each call to next yields a 3.

2. **(6.0 points)   Making a List, Checking it Twice**

Complete the environment diagram below and then answer the questions that follow. There is one question for each labeled blank in the diagram. The blanks with no labels have no questions associated with them and are not scored. **Some blanks may be empty or unused. If a blank contains an arrow to a function, write the function as it would appear in the diagram. Do not add frames for calls to built-in functions.**

Reminder: `list(s)` creates a new list with the same elements as `s`; `s.pop()` removes and returns the last element of `s`.

```
def push(t, x):
    s = list(t)
    s[1].append(x)

def pull():
    s = list(t)
    u = s.pop()
    return u[1][0]

s = [7, [8, 9]]
t = [6, s]

push(s, pull())
print(t)
```

```
Global    push  ──────────→ func push(t, x) [parent=Global]
          pull  ──────────→ func pull() [parent=Global]
          s     [  ]
          t     [  ]

f1: pull [parent=Global]
          s     [  ]
          u     (a)
          Return value  [  ]

f2: push [parent=Global]
          t     [  ]
          x     [  ]
          s     [  ]
          Return value   None
```

(a) **(3.0 pt)** What value fills blank (a)? Select **all** that apply.

☐ None

■ A list

☐ A number

■ The object bound to global `s`

☐ The object bound to global `t`

☐ The same object that `s[0]` would evaluate to in the global frame

☐ The same object that `s[1]` would evaluate to in the global frame

☐ The same object that `t[0]` would evaluate to in the global frame

■ The same object that `t[1]` would evaluate to in the global frame

(b) **(3.0 pt)** What would be printed by the expression `print(t)` at the end?

[6, [7, [8, 9, 8]]]

3. (6.0 points)   24-Hour Library

Anyone can check out a book from a library and then bring that book back. Implement the `Library` and `Book` classes. A `Library` takes a list of unique strings called `titles` and constructs a `books` dictionary that has strings as keys and `Book` objects as values. Its `checkout` method takes a string `title` and returns the corresponding `Book` object if that `title` is not checked out, or prints a message. After `bring_back` is invoked on that book, it can be checked out again.

```python
class Library:
    """A library with one copy of each title that can be checked out.

    >>> cs = Library(['Composing Programs', 'Python Docs', 'Berkeley Academic Guide'])
    >>> bs = [cs.checkout('Composing Programs'), cs.checkout('Python Docs')]
    >>> cs.checkout('Composing Programs')       # This time, no Book is returned
    Composing Programs is checked out
    >>> bs[0].bring_back()
    >>> cs.checkout('Composing Programs').title  # This time, a Book is returned
    'Composing Programs'
    """
    def __init__(self, titles):
        self.books = {t: Book(t, _____) for t in titles}
                                    (a)
        self.out = []  # A list of Book objects

    def checkout(self, title):
        assert title in self.books, title + " isn't in this library's collection"
        book = _____
                  (b)
        if book not in self.out:
            _____
              (c)
            return book
        else:
            print(book, 'is checked out')

class Book:
    def __init__(self, title, library):
        self.title = title      # a string
        self.library = library  # a Library object

    def bring_back(self):
        _____.remove(_____)
          (d)              (e)

    def __str__(self):
        return _____
                  (f)
```

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

  ● `self`

  ○ `self.books`

  ○ `Library`

  ○ `Library()`

  ○ `Library(titles)`

  The book takes a library. The libary that the book is in is the 'self" object that is being created with this dictionary of books.

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

  `self.books[title]`

  Get the relevant book from the dictionary of books.

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

  ○ `out.append(book)`

  ○ `out.extend(book)`

  ○ `Library.append(book)`

  ○ `Library.extend(book)`

  ● `self.out.append(book)`

  ○ `self.out.extend(book)`

  We wish to add the book to the list of checked out books. Append adds an element to a list. (Extend extends a list with another list.)

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

  ○ `self.`

  ○ `self.out`

  ○ `library`

  ○ `library.out`

  ● `self.library.out`

  Remove the book from the list of checked out books.

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

  ● `self`

  ○ `title`

  ○ `library`

  ○ `self.title`

  ○ `self.library`

  The name `self` is bound to the book we're removing.

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

○ `self`

○ `title`

● `self.title`

○ `repr(self)`

○ `repr(title)`

○ `repr(self.title)`

One needs to return a string with the title in it. The repr would return a string with quotes in it.

4. (16.0 points)   A Perfect Question

Definition. A *perfect square* is `k*k` for some integer `k`.

(a) (7.0 points)

Implement `fit`, which takes **positive** integers `total` and `n`. It returns `True` or `False` indicating whether there are **n positive** perfect squares that sum to `total`. The perfect squares need not be unique.

```
def fit(total, n):
    """Return whether there are n positive perfect squares that sums to total.

    >>> [fit(4, 1), fit(4, 2), fit(4, 3), fit(4, 4)]  # 1*(2*2) for n=1; 4*(1*1) for n=4
    [True, False, False, True]
    >>> [fit(12, n) for n in range(3, 8)]  # 3*(2*2), 3*(1*1)+3*3, 4*(1*1)+2*(2*2)
    [True, True, False, True, False]
    >>> [fit(32, 2), fit(32, 3), fit(32, 4), fit(32, 5)] # 2*(4*4), 3*(1*1)+2*2+5*5
    [True, False, False, True]
    """
    def f(total, n, k):

        if _____:
             (a)
            return True

        elif _____:
              (b)
            return False
        else:
            return _____
                      (c)

    return f(total, n, 1)
```

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

☐ `total == 0`

☐ `n == k`

■ `total == 0 and n == 0`

☐ `total == k * k`

■ `total == k * k and n == 1`

■ `total == n * k * k`

The final condition is ideal, but the other two would still provide the correct answer after additional recursive calls.

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

```
total < k * k
```

At this point, one can only use terms that are larger than `k*k` and thus there can't be a solution if total is smaller than that.

iii. (3.0 pt) Fill in blank (c) with an expression of the form `f(___, ___, ___) ___ f(___, ___, ___)`.

> `f(total - k * k, n - 1, k) or f(total, n, k + 1)`

You either use `k*k` at least once or you only use larger perfect squares.

(b) **(9.0 points)**

Implement the generator function `squares`, which takes **positive** integers `total` and `k`. It yields all lists of perfect squares greater or equal to `k*k` that sum to `total`. Each list is in non-increasing order (large to small).

```python
def squares(total, k):
    """Yield the ways in which perfect squares greater or equal to k*k sum to total.

    >>> list(squares(10, 1))  # All lists of perfect squares that sum to 10
    [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [4, 1, 1, 1, 1, 1, 1], [4, 4, 1, 1], [9, 1]]
    >>> list(squares(20, 2))  # Only use perfect squares greater or equal to 4 (2*2).
    [[4, 4, 4, 4, 4], [16, 4]]
    """
    assert total > 0 and k > 0
    if total == k * k:

        yield _____
                (d)
    elif total > k * k:

        for s in _____:
                   (e)

            yield _____
                    (f)
        yield from squares(total, k + 1)
```

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

```
[k * k]
```

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

```
squares(total - k * k, k)
```

This gives a generator that yields solutions whose total area is `total - k*k`. And adding one more square of `k*k` gives a solution for `total`.

iii. (1.0 pt) Select **all** of the options that could fill in blank (f).

☐ `s + k*k`

☐ `s.append(k*k)`

■ `s + [k*k]`

☐ `[s] + [k*k]`

Each `s` is a list whose elements add to `total - k*k`, and adding `k*k` gives a desired solution.

iv. **(3.0 pt)** Write an expression containing `a` that evaluates to the **shortest** list of perfect squares that sum to an integer `a`. For example, if `a = 32`, your expression should evaluate to `[16, 16]` (not `[25, 4, 1, 1, 1]`). If there are two or more such lists that are both shortest, it can evaluate to any of them. Assume `squares` is implemented correctly.

```
min(squares(a, 1), key=len)
```

5. **(7.0 points)   Only Paths**

Implement `only_paths`, which takes a Tree of numbers `t` and a number `n`. It returns a new Tree with only the nodes of `t` on a path from the root to a leaf with labels that sum to `n`, or `None` if no path sums to `n`. Do not mutate `t`.

The Tree class appears on the midterm 2 study guide. Here is an illustration of the doctest examples involving `t`.
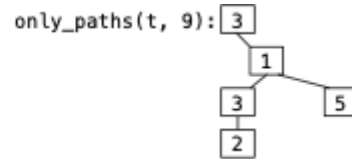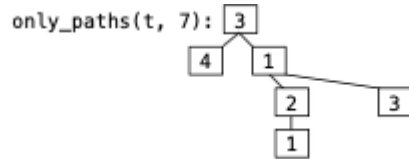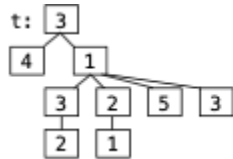

```
def only_paths(t, n):
    """Return a Tree with only the nodes of t along paths from the root to a leaf of t
    for which the node labels of the path sum to n. If no paths sum to n, return None.

    >>> print(only_paths(Tree(5, [Tree(2), Tree(1, [Tree(2)]), Tree(1, [Tree(1)])]), 7))
    5
      2
      1
        1
    >>> t = Tree(3, [Tree(4), Tree(1, [Tree(3, [Tree(2)]), Tree(2, [Tree(1)]), Tree(5), Tree(3)])])
    >>> print(only_paths(t, 7))
    3
      4
      1
        2
          1
        3
    >>> print(only_paths(t, 9))
    3
      1
        3
          2
        5
    >>> print(only_paths(t, 3))
    None
    """
    if _____:
           (a)

        return t

    new_branches = [_____ for b in t.branches]
                       (b)

    if _____(new_branches):
          (c)

        return Tree(t.label, [b for b in new_branches if _____])
                                                           (d)
```

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

```
t.is_leaf() and n == t.label
```

This is a tree with a path to a leaf with sum of labels being n as specified. And no recursive call is needed.

(b) (3.0 pt) Fill in blank (b).

```
only_paths(b, n - t.label)
```

The sum of labels along a path to any leaf is the sum in a branch plus the label. Thus, we wish to find a solution for each branch with sum `n-t.label`.

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

○ `all`

● `any`

○ `has`

○ `only_paths`

The function returns `None` if a branch has no valid path. Thus, if all the entries of the list of solutions for branches are `None`, the procedure is done. Otherwise it should return a tree.

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

○ `only_paths(b, n)`

○ `b.label == n`

○ `b.label != n`

○ `b is None`

● `b is not None`

Only keep the results that are trees, not `None`.

(e) **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!** Fill in the blank to implement `only_long_paths`, which takes a Tree of numbers `t` and a number `n`. It returns a new Tree containing only the nodes of `t` that lie on a path from the root to a leaf for which the sum of the labels **plus the length of the path** is n. Do not mutate `t`. You may not write `and`, `or`, `if`, `[` or `]`. Assume `only_paths` is implemented correctly.

```
only_paths = _____

def only_long_paths(t, n):
    """Return a Tree with only the nodes of t along paths from the root to a leaf of t
    for which the sum of node labels plus the length of the path is n.

    >>> example = Tree(5, [Tree(3), Tree(1, [Tree(2)]), Tree(1, [Tree(1)])])
    >>> only_long_paths(example, 10)  # Result has paths 5-3 (length 2) and 5-1-1 (length 3)
    Tree(5, [Tree(3), Tree(1, [Tree(1)])])
    """
    return only_paths(t, n)
```

```
(lambda f: lambda t, n: f(t, n-1))(only_paths)
```

Like `memo` from lecture, this lambda function takes a function and returns a similar function. This similar function calls the original `only_paths` on `n-1` rather than `n` to account for the length of the path.

6. **(6.0 points)   After Party**

Implement `after`, which takes a linked list `s` and values `a` and `b`. It returns whether an element of `s` equal to `b` appears after an element of `s` equal to `a`. The `Link` class appears on the Midterm 2 study guide.

```
def after(s, a, b):
    """Return whether b comes after a in linked list s.

    >>> t = Link(3, Link(6, Link(5, Link(4))))
    >>> after(t, 6, 4)
    True
    >>> after(t, 4, 6)
    False
    >>> after(t, 6, 6)
    False
    """
    def find(s, n, f):

        if s == Link.empty:

            return _____
                     (a)

        elif s.first == n:
            return f(s.rest)
        else:
            return find(s.rest, n, f)

    return find(s, a, lambda rest: _____)
                                      (b)
```

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

○ n

○ True

● False

○ a < b

○ f(Link.empty)

You have not found n, which is a, yet, and the list is done.

(b) **(4.0 pt)** Fill in blank (b). **You may not use or, and, if, [, or].**

```
find(rest, b, lambda rest: True)
```

At this point, you have found a in the list, so you should see if b is in the rest of the list.

(c) **(1.0 pt)** What is the order of growth of the run time of an efficient implementation of `after` in terms of the length of `s`? Here, *efficient* means that you have filled in the blanks to give the function the fastest order of growth.

○ Constant

○ Logarithmic

● Linear

○ Quadratic

○ Exponential

Each element is considered once since each recursive function call is called on the rest of the current list, giving linear time. In the best case, `a` and `b` could appear at the beginning of the list, which would be found in constant time, so that answer was accepted as well.

No more questions.