CS 61A Fall 2023

Structure and Interpretation of Computer Programs

Midterm 2

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*

(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*

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

- ◯ `3`
- ◯ `5`
- ◯ `range(3, 7)`
- ◯ `range(5, 7)`
- ◯ *Error*

(d) (1.0 pt) `next(iter(s)) + next(iter(s))`

- ◯ `6`
- ◯ `7`
- ◯ `10`
- ◯ `11`
- ◯ *Error*

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`.

nYW8jSG.png

(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?

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.

```
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)`

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

(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)`

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

○ `self.`

○ `self.out`

○ `library`

○ `library.out`

○ `self.library.out`

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

○ `self`

○ `title`

○ `library`

○ `self.title`

○ `self.library`

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

- ◯ `self`
- ◯ `title`
- ◯ `self.title`
- ◯ `repr(self)`
- ◯ `repr(title)`
- ◯ `repr(self.title)`

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`

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

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

(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).

```
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).

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

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]`

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.

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`.

L1ifGv3.png

```
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).

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

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

- ○ `all`
- ○ `any`
- ○ `has`
- ○ `only_paths`

(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`

(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)
```

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)`

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

(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

No more questions.