# CS 61A Structure and Interpretation of Computer Programs
# Spring 2023

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

Assume the following code has been executed.

```
def chain(s):
    return [s[0], s[1:]]

silver = [2, chain([3, 4, 5])]
gold = [silver[0], silver[1].pop()]
silver[0] = 1
platinum = chain(chain([6, 7, 8]))
```

Write the output **displayed by the interactive Python interpreter** when each expression below is evaluated.

*Hint*: Try drawing an environment diagram!

*Reminder*: `s.pop()` removes and returns the last item in list `s`.

**(a) (2.0 pt)** `silver`

**(b) (2.0 pt)** `gold`

**(c) (2.0 pt)** `platinum`

**2. (10.0 points) Letter Grade**

**(a) (4.0 points)**

Implement the `Letter` class. A `Letter` has two instance attributes: `contents` (a `str`) and `sent` (a `bool`). Each `Letter` can only be sent once. The `send` method prints whether the letter was sent, and if it was, returns the reply, which is a new `Letter` instance with the same `contents`, but in all caps. *Hint*: `'hi'.upper()` evaluates to `'HI'`.

```
class Letter:
    """A letter receives an all-caps reply when sent successfully.

    >>> hi = Letter('Hello, World!')
    >>> hi.send()
    Hello, World! has been sent.
    HELLO, WORLD!
    >>> hi.send()
    Hello, World! was already sent.
    >>> Letter('Hey').send().send()
    Hey has been sent.
    HEY has been sent.
    HEY
    """
    def __init__(self, contents):
        self.contents = contents
        _________
           (a)
    def send(self):
        if self.sent:
            print(self, 'was already sent.')
        else:
            print(self, 'has been sent.')
            _________
               (b)
            return _________
                       (c)
    def __repr__(self):
        # Note: since no __str__ method is defined, the repr and str strings are the same.
        return self.contents
```

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

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

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

**(b) (6.0 points)**

Implement the `Numbered` class. A `Numbered` letter has a `number` attribute equal to how many numbered letters have previously been constructed. This number appears in its `repr` string. Assume `Letter` is implemented correctly.

```
class Numbered(Letter):
    """A numbered letter has a different repr method that shows its number.

    >>> hey = Numbered('Hello, World!')
    >>> hey.send()
    #0 has been sent.
    HELLO, WORLD!
    >>> Numbered('Hi!').send()
    #1 has been sent.
    HI!
    >>> hey
    #0
    """
    number = 0

    def __init__(self, contents):
        super().__init__(contents)
        _________
           (d)
        _________
           (e)

    def __repr__(self):
        return '#' + _________
                        (f)
```

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

- ◯ `Numbered.number = 0`
- ◯ `Numbered.number = number`
- ◯ `Numbered.number = self.number`
- ◯ `self.number = 0`
- ◯ `self.number = number`
- ◯ `self.number = Numbered.number`

**ii. (2.0 pt)** Fill in blank (e).

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

**3. (9.0 points) Prefixes**

**Definition.** A *prefix sum* of a sequence of numbers is the sum of the first `n` elements for some positive length `n`.

**(a) (4.0 points)**

Implement `prefix`, which takes a list of numbers `s` and returns a list of the prefix sums of `s` in increasing order of the length of the prefix.

```
def prefix(s):
    """Return a list of all prefix sums of list s.

    >>> prefix([1, 2, 3, 0, 4, 5])
    [1, 3, 6, 6, 10, 15]
    >>> prefix([2, 2, 2, 0, -5, 5])
    [2, 4, 6, 6, 1, 6]
    """
    return [_________ for k in _________]
               (a)                (b)
```

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

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

- ◯ `s`
- ◯ `[s]`
- ◯ `s[1:]`
- ◯ `range(s)`
- ◯ `range(len(s))`

**iii. (1.0 pt)** What is the order of growth of the time to run this alternate implentation in terms of the length of `s`?

Assume that the `append` method of a list takes just 1 step (constant time).

```
def prefix(s):
    """Return a list of all prefix sums of list s."""
    t = 0
    result = []
    for x in s:
        t = t + x
        result.append(t)
    return result
```

- ◯ Constant
- ◯ Logarithmic
- ◯ Linear
- ◯ Quadratic
- ◯ Exponential

**(b) (5.0 points)**

Implement `tens`, which takes a **non-empty** linked list of numbers `s` represented as a `Link` instance. It prints all of the prefix sums of `s` that are multiples of 10 in increasing order of the length of the prefix.

The `Link` class appears on the midterm 2 study guide.

```
def tens(s):
    """Print all prefix sums of Link s that are multiples of ten.

    >>> tens(Link(3, Link(9, Link(8, Link(10, Link(0, Link(14, Link(6)))))))
    20
    30
    30
    50
    """

    def f(suffix, total):

        if total % 10 == 0:

            print(total)

        if _________:
              (d)

            _________
               (e)

    _________
       (f)
```

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

- ◯ `s is not Link.empty`
- ◯ `s.rest is not Link.empty`
- ◯ `s.rest.rest is not Link.empty`
- ◯ `suffix is not Link.empty`
- ◯ `suffix.rest is not Link.empty`
- ◯ `suffix.rest.rest is not Link.empty`

**ii. (2.0 pt)** Fill in blank (e).

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

**4. (10.0 points) Tree Trimming**

**(a) (7.0 points)**

Implement `exclude`, which takes a `Tree` instance `t` and a value `x`. It returns a `Tree` containing the root node of `t` as well as each non-root node of `t` with a label not equal to `x`. The parent of a node in the result is its nearest ancestor node that is not excluded. The input `t` must not be modified.

The `Tree` class appears on the midterm 2 study guide.

```
def exclude(t, x):
    """Return a Tree with the non-root nodes of t whose labels are not equal to x.

    >>> t = Tree(1, [Tree(2, [Tree(2), Tree(3)]), Tree(4, [Tree(1)])])
    >>> exclude(t, 2)
    Tree(1, [Tree(3), Tree(4, [Tree(1)])])
    >>> t              # t is not changed
    Tree(1, [Tree(2, [Tree(2), Tree(3)]), Tree(4, [Tree(1)])])
    >>> exclude(t, 1)  # The root node cannot be excluded
    Tree(1, [Tree(2, [Tree(2), Tree(3)]), Tree(4)])
    """

    filtered_branches = map(lambda y: _________, t.branches)
                                          (a)

    bs = []

    for b in filtered_branches:

        if _________:
              (b)

            bs._________(_________)
                  (c)       (d)

        else:

            bs.append(b)

    return Tree(t.label, bs)
```

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

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

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

- ◯ `remove`
- ◯ `pop`
- ◯ `append`
- ◯ `extend`

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

- ◯ `b`
- ◯ `b.branches`
- ◯ `exclude(b, x)`
- ◯ `Tree(b.label, [exclude(y, x) for y in b.branches])`

**(b) (3.0 points)**

Implement `remove`, which takes a `Tree` instance `t` and a value `x`. It removes all non-root nodes from `t` that have a label equal to `x`, then returns `t`. The parent of a node in `t` is its nearest ancestor that is not removed. Assume that `exclude` is implemented correctly.

```
def remove(t, x):
    """Remove all non-root nodes labeled x from t and return t.

    >>> t = Tree(1, [Tree(2, [Tree(2), Tree(3)]), Tree(4)])
    >>> remove(t, 2)
    Tree(1, [Tree(3), Tree(4)])
    >>> remove(t, 3)  # t was changed, so both 2 and 3 are now removed.
    Tree(1, [Tree(4)])
    """

    ---------
       (e)

    return t
```

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

**5. (10.0 points) Parking**

**Definition.** When parking vehicles in a row, a *motorcycle* takes up 1 parking spot and a *car* takes up 2 adjacent parking spots. A string of length `n` can represent `n` adjacent parking spots using `%` for a motorcycle, `<>` for a car, and `.` for an empty spot. For example `'.%%.<><>'` represents an empty spot, then two spots containing motorcycles, then another empty spot, then four spots containing two cars.

Thanks to the Berkeley Math Circle for introducing this question.

**(a) (4.0 points)**

Implement `count_park`, which returns the number of ways that vehicles can be parked in `n` adjacent parking spots for positive integer `n`. Some or all spots can be empty.

```
def count_park(n):
    """Count the ways to park cars and motorcycles in n adjacent spots.

    >>> count_park(1)  # '.' or '%'
    2
    >>> count_park(2)  # '..', '.%', '%.', '%%', or '<>'
    5
    >>> count_park(4)  # some examples: '<><>', '.%%.', '%<>%', '%.<>'
    29
    """

    if n < 0:

        return 0

    if n == 0:

        return __________
                   (a)

    return __________
              (b)
```

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

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

**(b) (6.0 points)**

Implement `park`, a generator function that takes a non-negative integer `n`. It yields (in any order) all possible strings representing `n` adjacent parking spots.

```
def park(n):
    """Yield the ways to park cars and motorcycles in n adjacent spots.

    >>> sorted(park(1))
    ['%', '.']
    >>> sorted(park(2))
    ['%%', '%.', '.%', '..', '<>']
    >>> sorted(park(3))
    ['%%%', '%%.', '%.%', '%..', '%<>', '.%%', '.%.', '..%', '...', '.<>', '<>%', '<>.']
    >>> len(list(park(4)))
    29
    """
    if n == 0:
        yield ''
    elif n > 0:
        for s in park(n-1):
            ---------
               (c)
            ---------
               (d)

        for s in _________:
                     (e)
            ---------
               (f)
```

**i. (3.0 pt)** Fill in blanks (c) and (d) with two lines of code. You may **not** use `yield from`.

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

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

**(c) 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!**

Implement `pack`, a generator function that takes a non-negative integer `n`. It yields (in any order) all possible strings representing `n` adjacent parking spots in which an equal number of cars and motorcycles are parked, no motorcycle is adjacent to a car, and no empty spot is adjacent to another empty spot.

```
def pack(n):
    """Yield the ways to park an equal number of cars and motorcycles
    in n adjacent spots with no motorcycle adjacent to a car and no
    empty spot adjacent to another empty spot.

    >>> sorted(pack(4))
    ['%.<>', '<>.%']
    >>> sorted(pack(7))
    ['%%.<><>', '<><>.%%']
    >>> sorted(pack(8))
    ['%%.<>.<>', '%%.<><>.', '%.%.<><>', '%.<><>.%', '.%%.<><>',
     '.<><>.%%', '<>.%%.<>', '<>.<>.%%', '<><>.%%.', '<><>.%.%']
    """
    def f(n, k):
        if n == 0 and k == 0:
            yield ''
        elif n > 0:
            yield from g(n-1, k-1, '<', '%' )
            yield from g(n-1, k,   '.', '.' )
            yield from g(n-2, k+1, '%', '<>')

    def g(n, k, no, yes):

        yield from _________

    yield from f(n, 0)
```

**i.** Fill in the blank.

**No more questions.**