# CS 61A Structure and Interpretation of Computer Programs
# Summer 2019

PRACTICE FINAL

**INSTRUCTIONS**

- You have 3 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except three hand-written 8.5" × 11" crib sheet of your own creation and the official CS 61A midterm and final study guides.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

| | |
|---|---|
| Last name | Xia |
| First name | Sabrina |
| Student ID number | 3032064101 |
| CalCentral email (`_@berkeley.edu`) | |
| TA | |
| Exam Room | |
| Name of the person to your left | |
| Name of the person to your right | |
| *All the work on this exam is my own.* **(please sign)** | |

**POLICIES & CLARIFICATIONS**

- If you need to use the restroom, bring your phone and exam to the front of the room.
- You may use built-in Python functions that do not require import, such as `min`, `max`, `pow`, `len`, `abs`, `sum`, `next`, `iter`, `list`, `tuple`, `map`, `filter`, `zip`, `all`, and `any`.
- You **may not** use example functions defined on your study guides unless a problem clearly states you can.
- For fill-in-the-blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.
- You may use the `Tree` and `Link` classes defined on Page 4 (left column) of the Study Guide.

**1. (12 points) Class Hierarchy**

For each row below, write the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and **expressions may affect later expressions**.

Whenever the interpreter would report an error, write Error. You *should* include any lines displayed before an error. *Reminder*: The interactive interpreter displays the `repr` string of the value of a successfully evaluated expression, unless it is `None`. Assume that you have started Python 3 and executed the following:

```
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'My job is to gather wealth'
class Proletariat(Worker):
    greeting = 'Comrade'
    def work(self, other):
        other.greeting = self.greeting + ' ' + other.greeting
        other.work() # for revolution
        return other
jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

| Expression | Interactive Output |
|---|---|
| `5*5` | 25 |
| `1/0` | Error |
| `Worker().work()` | 'Sir, I work' |
| `jack` | Peon |
| `jack.work()` | 'Maam, I work' |

| Expression | Interactive Output |
|---|---|
| `john.work()[10:]` | Peon, I work to gather wealth' |
| `Proletariat().work(john)` | Comrade Peon, I work<br>~~My job is to gather~~ wealth'<br>Peon |
| `john.elf.work(john)` | 'Comrade Peon, I work' |

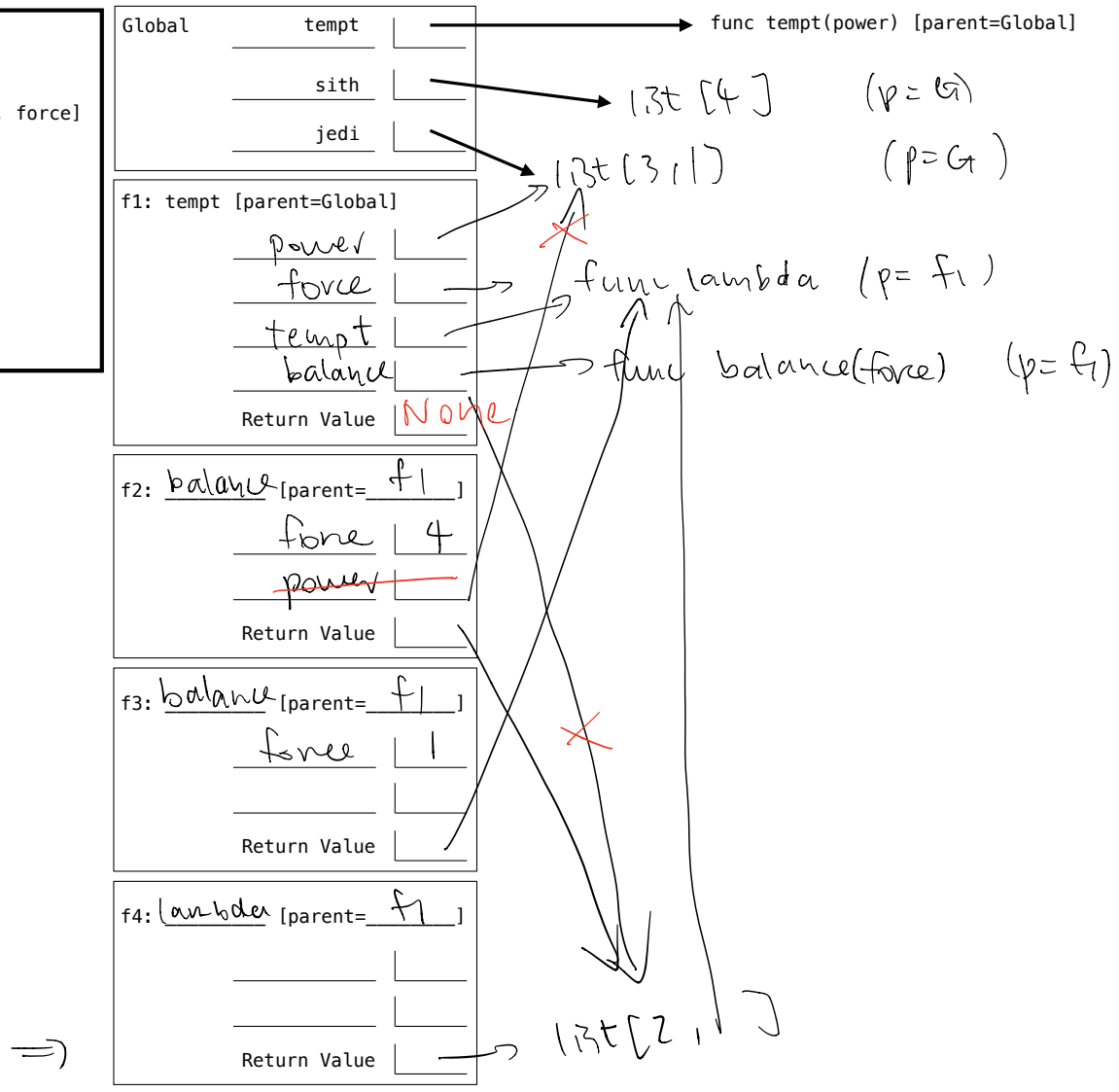
**2. (8 points) Endor**

Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to all frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
1  sith = []
2  jedi = [3]
3
4  def tempt(power):
5      force = lambda: [len(power), force]
6      tempt = force
7      def balance(force):
8          nonlocal power
9          power.append(force)
10         if force < 2:
11             return tempt
12         power = jedi
13         return balance(1)()
14     balance(4)
15
16 tempt(sith)
```



**3. (8 points) Forest Path**

**Definition.** A *path* through a `Tree` is a list of adjacent node values that starts with the root value and ends with a leaf value. For example, the paths of `Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])` are

```
[1, 2]
[1, 3, 4]
[1, 3, 5]
```

The `Tree` class is defined on the midterm 2 study guide. The `one` function defined below is used in the questions below to convert true and false values into the numbers 1 and 0, respectively.

```
def one(b):
    if b:
        return 1
    else:
        return 0
```

**(a) (3 pt)** Implement `bigpath`, which takes a `Tree` instance `t` and an integer `n`. It returns the number of paths in `t` whose sum is *at least* `n`. Assume that all node values of `t` are integers.

```
def bigpath(t, n):
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> bigpath(t, 3)
    3
    >>> bigpath(t, 6)
    2
    >>> bigpath(t, 9)
    1
    """


    if t.is_leaf():

        return one(t.label >= n)

    return sum([big_path(b, n-t.label) for b in t.branches])
```

**Definition.** A *path* through a `Tree` is a list of adjacent node values that starts with the root value and ends with a leaf value. For example, the paths of `Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])` are

```
[1, 2]
[1, 3, 4]
[1, 3, 5]
```

**(b) (3 pt)** Implement `allpath` which takes a `Tree` instance `t`, a one-argument predicate `f`, a two-argument reducing function `g`, and a starting value `s`. It returns the number of paths `p` in `t` for which `f(reduce(g, p, s))` returns a true value. The `reduce` function is on the final study guide. You do not need to call it, though.

```
def allpath(t, f, g, s):
    """Return the number of paths p in t for which f(reduce(g, p, s)) is true.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> even = lambda x: x % 2 == 0
    >>> allpath(t, even, max, 0)  # Path maxes are 2, 4, and 5; 2 & 4 are even
    2
    >>> allpath(t, even, pow, 2)  # E.g., pow(pow(2, 1), 2) is even
    3
    >>> allpath(t, even, pow, 1)  # Raising 1 to any power is odd
    0
    """


    if t.is_leaf():

        return one(_____)

    return sum([_____])
```

**(c) (2 pt)** Re-implement `bigpath` (part a) using `allpath` (part c). Assume `allpath` is defined correctly.

```
from operator import add, mul

def bigpath(t, n):
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> bigpath(t, 3)
    3
    >>> bigpath(t, 6)
    2
    >>> bigpath(t, 9)
    1
    """

    return allpath(t, _____, _____, _____)
```

**4. (8 points) Cucumber**

Cucumber is a card game. Cards are positive integers (no suits). Players are numbered from 0 up to `players` (0, 1, 2, 3 in a 4-player game). In each `Round`, the players each `play` one card, starting with the `starter` and in ascending order (player 0 follows player 3 in a 4-player game). If the `card` played is as high or higher than the `highest` card played so far, that player takes `control`. The winner is the last player who took control after every player has played once. Implement `Round` so that `play_round` behaves as described in the doctests below. Part of your score on this question will be assigned based on *composition* (don't repeat yourself).

```
def play_round(starter, cards):
    """Play a round and return all winners so far. Cards is a list of pairs.
    Each (who, card) pair in cards indicates who plays and what card they play.

    >>> play_round(3, [(3, 4), (0, 8), (1, 8), (2, 5)])
    [1]
    >>> play_round(1, [(3, 5), (1, 4), (2, 5), (0, 8), (3, 7), (0, 6), (1, 7)])
    It's not your turn, player 3
    It's not your turn, player 0
    The round is over, player 1
    [1, 3]
    >>> play_round(3, [(3, 7), (2, 5), (0, 9)]) # Round is never completed
    It's not your turn, player 2
    [1, 3]
    """
    r = Round(starter)
    for who, card in cards:
        try:
            r.play(who, card)
        except AssertionError as e:
            print(e)
    return Round.winners

class Round:
    players, winners = 4, []
    def __init__(self, starter):
        self.starter, self.player, self.highest = starter, starter, -1

    def play(self, who, card):
        assert ____not self.complete()____, 'The round is over, player '+str(who)
        assert ____self.player == who____, "It's not your turn, player "+str(who)
        self.player = ____(who+1) % self.players____

        if card >= self.highest:
            ____self.highest____, ____self.control____ = ____card____, ____who____
        if ____self.complete()____:
            self.winners.append(self.control)

    def complete(self):
        return ____self.player == self.starter and self.highest != -1____
```

**5. (14 points) Grouper**

**(a) (4 pt)** Implement `group`, which takes a one-argument function `f` and a list `s`. It returns a list of groups. Each group is a list that contains all the elements `x` in `s` that return equal values for `f(x)`. The elements in a group appear in the same order that they appeared in `s`. The groups are ordered by the order in which their first elements appeared in `s`.

```
def group(f, s):
    """Return a list of groups that contain all x with equal f(x).
    >>> five = [3, 4, 5, 2, 1]
    >>> group(lambda x: x % 2, five)
    [[3, 5, 1], [4, 2]]
    >>> group(lambda x: x % 3, five)
    [[3], [4, 1], [5, 2]]
    """
    a = []
    for b in map(f, s):
        if ____b not in a____:
            a.append(b)
    return [[__x for x in s if f(x)==b__] for b in a]
```

**(b) (4 pt)** Implement `group_link`, which takes a one-argument function `f` and a `Link` instance `s`. It returns a linked list of groups. Each group is a `Link` instance containing all the elements `x` in `s` that return equal values for `f(x)`. The order of groups and elements is the same as for `group`. The `Link` class appears on your midterm 2 study guide. The `filter_link` function appears in the appendix on the last page of this exam.

```
def group_link(f, s):
    """Return a linked list of groups that contain all x with equal f(x).

    >>> five = Link(3, Link(4, Link(5, Link(2, Link(1)))))
    >>> group_link(lambda x: x % 2, five)
    Link(Link(3, Link(5, Link(1))), Link(Link(4, Link(2))))
    >>> group_link(lambda x: x % 3, five)
    Link(Link(3), Link(Link(4, Link(1)), Link(Link(5, Link(2)))))
    """
    if s is Link.empty:
        return s
    else:
        a = filter_link(lambda x: ___x==f(s.first)___, ____s____)
        b = filter_link(lambda x: ___x!=f(s.first)___, ____s____)
        return Link(a, group_link(f, b))
```

**Definition.** The *multi-grouping* by function `f` of list `s` is formed by the following iterative process with `k` starting at 0 and increasing by 1 each iteration.

- Group together all elements that yield equal values when applying `f` repeatedly, `k` times.
- If all elements are in a single group, the process is complete. Otherwise, place each new group in a (possibly nested) list and repeat.

For example, if `f` is `lambda x: max(x-3, 0)` and `s` is `[2, 4, 3, 4, 2]`, then

- In the `k=0` iteration, the 2's are grouped, the 4's are grouped, and the 3 is alone: `[[2, 2], [3], [4, 4]]`
- In the `k=1` iteration, `f(2)==f(3)`, and so the 2's group and 3's group are grouped: `[[[2, 2], [3]], [[4, 4]]]`
- In the `k=2` iteration, `f(f(2))==f(f(3))==f(f(4))`. All elements are in a single group, so we're done.

**(c) (4 pt)** Implement `multigroup`, which returns the multi-grouping by `f` of a list `s`. Assume that the process terminates and that `group` (part a) is implemented correctly.

```
def multigroup(f, s):
    """Return a multi-grouping by f of the elements in s.

    >>> multigroup(lambda x: max(x-3, 0), [2, 4, 3, 4, 2])
    [[[2, 2], [3]], [[4, 4]]]
    >>> multigroup(abs, [5])
    5
    >>> multigroup(abs, [5, 5])
    [5, 5]
    >>> multigroup(abs, [5, 5, -5])
    [[5, 5], [-5]]
    >>> multigroup(lambda x: x // 10, [123, 145, 126, 149])
    [[[123], [126]], [[145], [149]]]
    >>> multigroup(lambda x: x[1:], ['tin', 'man', 'can'])
    [[['tin']], [['man'], ['can']]]
    >>> multigroup(lambda x: max(x-1, 0), [2, 4, 3, 4, 2])
    [[[[2, 2]]], [[[3]]], [[[4, 4]]]]
    """
    def using(g, s):

        if len(s) == 1:

            return s[0]

        else:

            grouped = group(g, s)

            return using(lambda x: f(g(x[0])), grouped)

    return using(lambda x: x, s)
```

**(d) (2 pt)** How many square brackets are in the return value of `multigroup(hail, [3, 20, 128])`? Assume that `multigroup` is implemented correctly.

```
def hail(x):
    if x == 1:
        return 1
    elif x % 2 == 0:
        return x // 2
    else:
        return 3 * x + 1
```

[ [3] [20] [128] ]
[[ [3] [20] ] [ [128] ]]
[[[ [3] [20] ] [ [128] ]]]
10 10 64
5 5 32

16

**6. (6 points) Don't repeat yourself**

**(a) (2 pt)** Implement `repeater`, which takes in a list of positive numbers and returns a list where every number in the original list except for the first number appears a number of times equivalent to the previous number.

```
scm> (repeater nil)
()
scm> (repeater '(1 2 3))
(2 3 3)
scm> (repeater '(4 1 2 5))
(1 1 1 1 2 5 5)
```

```
(define (repeater nums)
    (define (repeat nums n)
        (cond ((null? nums) nil)
              ((= n 0) (repeat (cdr nums) (car nums)))
              (else (cons (car nums) (repeat nums (- n 1))))))
    (repeat (cdr nums) (car nums)))
```

**(b) (4 pt)** Implement `zip-tail`, which is a tail recursive procedure that takes in two lists `a` and `b` and returns a single list containing two-element lists of co-indexed elements from `a` and `b`. If one list is shorter than the other, the zipped list's length is that of the shorter list. Your solution should be tail recursive.

```
scm> (zip-tail '(1 2 3) '(4 5 6))
((1 4) (2 5) (3 6))
scm> (zip-tail '(c 6 a) '(s 1 ! hello world))
((c s) (6 1) (a !))
```

You may use the built-in `append` procedure, which you can assume is tail recursive.

```
scm> (append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
```

```
(define (zip-tail a b)
    (define (zipper a b result)
        (if (or (null? a) (null? b)) result
            (zipper (cdr a) (cdr b) (append result
                (list (list (car a) (car b)))))))
    (zipper a b '()))
```

**7. (8 points) Highly Exclusive**

**(a) (4 pt)** Complete the definition of `no-fib`, the stream of all positive integers that are not Fibonacci numbers. These are all positive integers excluding 1, 2, 3, 5, 8, 13, ... The stream starts with 4, 6, 7, 9, 10, 11, 12, 14.

```
(define (p prev curr n)
  (if (= n curr)
      (p curr (+ prev curr) (+ n 1))
      (cons-stream n (p prev curr (+ n 1)))))

(define no-fib (p 3 5 4))
```

**(b) (2 pt)** When the Berts eat at a restaurant, they record a review in a SQL table called `reviews`:

| restaurant | user | stars | review |
|---|---|---|---|
| Barney's | Albert | 4 | Used to like it |
| Chipotle | Robert | 5 | BOGO! BOGO! |
| Eureka | Albert | 5 | My favorite! |
| Bongo Burger | Albert | 2 | When I'm desperate |
| Umami Burger | Albert | 5 | I love truffle fries! |

Write an SQL query to figure out how many restaurants Albert gave 4 or 5 stars. Using the table above, the output to your query should be the following:

| stars | number of reviews |
|---|---|
| 4 | 1 |
| 5 | 2 |

```
select stars, count(*) from reviews where user = 'Albert'
  group by stars having stars >= 4;
```

**(c) (4 pt)** Select all positive integers that have at least 3 proper multiples that are less than or equal to $X$. A proper multiple $m$ of $n$ is an integer larger than $n$ such that $n$ evenly divides $m$ (`m % n == 0`).

The resulting table should have two columns. Each row contains an integer (that has at least 3 proper multiples) and the number of its proper multiples up to X. For example, the integer 3 has 5 proper multiples up to 20: 6, 9, 12, 15, and 18. Therefore, `3|5` is a row. There are five rows in the table when X is 20: `1|19`, `2|9`, `3|5`, `4|4`, and `5|3`. Your statement must work correctly even if $X$ changes.

We have provided two tables below: the seocnd table, *ints*, contains all integers between 1 and $X$ (inclusive). You do ***not*** need to understand how *ints* was created - just assume it contains the correct values.

```
create table X as select 20 as X;
with ints(n) as (select 1 union select n+1 from ints, X where n < X)

select a.n, count(*) from ints as a, ints as b
  where (b.n % a.n) = 0 AND b.n > a.n
  group by a.n having count(*) >= 3;
```

**Appendix.** *This is not a question.*

```
def filter_link(f, s):
    """Return a Link with the elements of s for which f returns a true value."""
    if s is Link.empty:
        return s
    else:
        filtered = filter_link(f, s.rest)
        if f(s.first):
            return Link(s.first, filtered)
        else:
            return filtered
```

# Post-Exam Reflection

Which problem type did you lose the majority of points on?
(e.g. What Would Python Print, Environment Diagram, Coding, etc.)

Coding

**For each problem** that you did not receive full credit for, please answer the following questions (You can either answer for an entire problem, or for each problem subsection individually- it's up to you. Under each prompt, you can list your responses for each problem consecutively.):

1. After looking over the solution can you explain why the solution is correct?

1. No. Have to look at walk-through video
2. No
4. Yes
5. Yes

2. Are any parts of your original answer on the right track? If so, which parts?

1. Two lines are correct. My logic and steps in evaluating the expression is on the right track, but details are wrong a few.
2. Steps in evaluating correct. Syntax (e.g. list vs diagram) is wrong.
4. Structure is correct. self.player → I know what this supposed to do, but don't know how to implement it. complete() → Stuck on how to distinguish start from completed
5. Coding is so hard! Also didn't have enough time to go through doctests & find commonality.

3. What led you to lose points on this problem? (In other words, what did you get stuck on or what did you miss?)

Answered in previous question!

4. What are key ideas here that could be helpful in future problems?

Don't leave anything blank- always write down what I have in mind. Most of the time it's very close to the correct answer

Spend more time on the prompt, doctests and the code given. understanding