CS 61A
Fall 2020

Sample Exam.

Final Exam

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 `albertxu3@berkeley.edu`. 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?

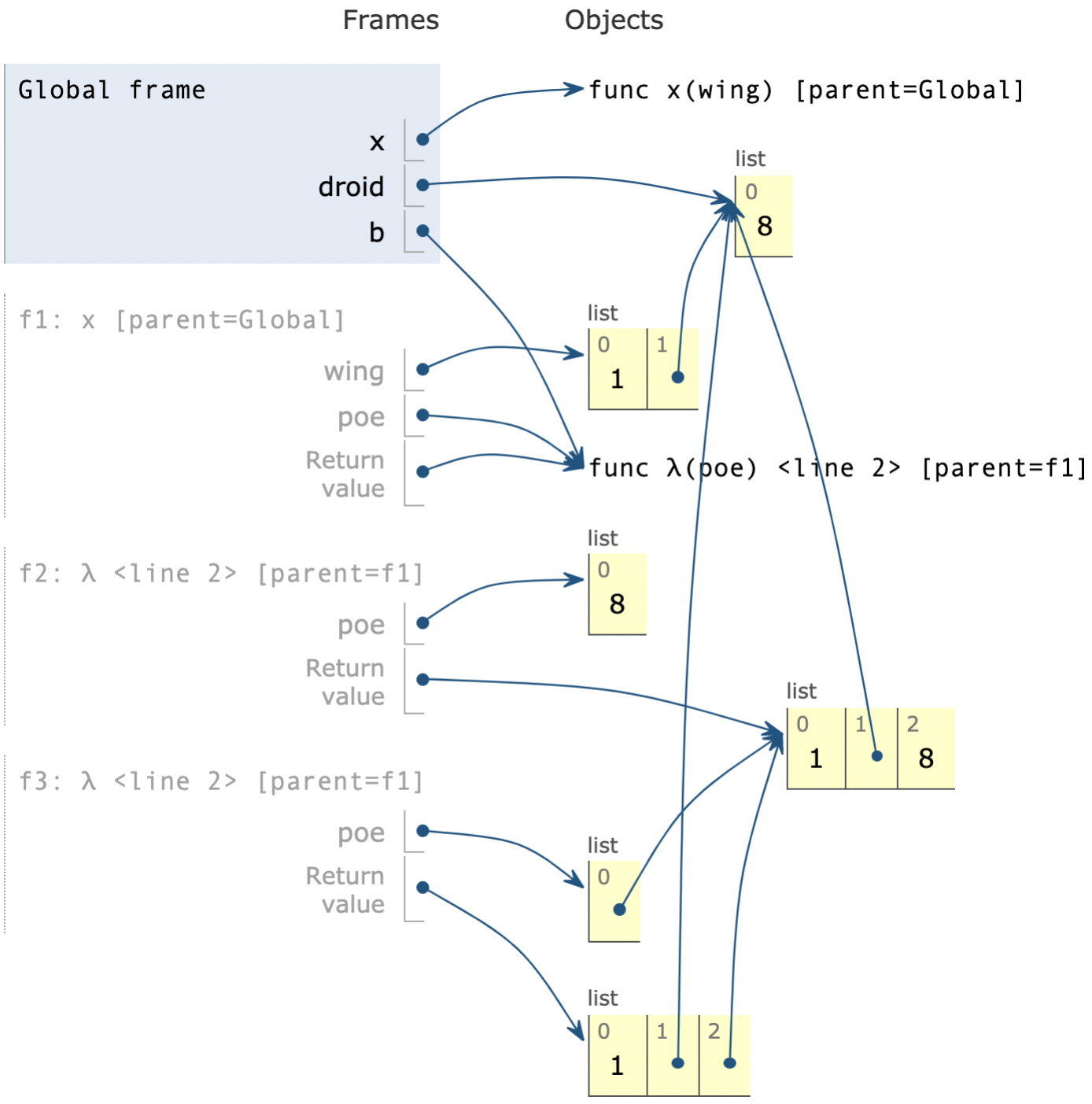
(b) What is your student ID number?

1. (6 points) The Droids You're Looking For

Fill in each blank in the code example below so that executing it would generate the following environment diagram on tutor.cs61a.org.

RESTRICTIONS. You must use all of the blanks. Each blank can only include one statement or expression.

Click here to open the diagram in a new window/tab




```
def x(wing):

    poe = lambda poe: _________
```

```
    #                    (a)

    wing.append(_________)
    #               (b)

    return _________
    #         (c)

droid = [8]

b = x([1])

---------
#  (d)
```

(a) **(1 pt)** Which of these could fill in blank (a)? Check all that apply.

- [x] `list(wing).extend(poe)` ✗
- [ ] `list(wing).append(poe)`
- [ ] `wing.append(poe)`
- [ ] `wing.extend(poe)`
- [x] `wing + poe` ✓

(b) **(1 pt)** Fill in blank (b).

droid

(c) **(1 pt)** Which of these could fill in blank (c)?

- ○ `poe(droid)`
- ○ `poe(b)`
- ○ `poe(wing)`
- ● `poe`

(d) **(3 pt)** Fill in blank (d).

b([b(list(droid))])

2. (16 points) Stoned

Definition: A *hailstone sequence* begins with a positive integer `n`. If `n` is even, divide it by 2. If `n` is odd, triple it and add 1. Repeat until 1 is reached. For example, the hailstone sequence starting at 10 is 10, 5, 16, 8, 4, 2, 1.

Assume that all hailstone sequences are finite.

(a) (9 points)

Implement `hailstone`, which takes a positive integer `n` and a one-argument function `g`. It calls `g` on each element of the hailstone sequence starting at `n` and returns the length of the sequence.

```
def hailstone(n, g):
    """Call g on each element of the hailstone sequence starting
    at n and return its length.

    >>> a = hailstone(10, print)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    >>> s = []
    >>> hailstone(10, s.append)
    7
    >>> s
    [10, 5, 16, 8, 4, 2, 1]
    """
    if n == 1:

        h = _________
        #      (a)

    elif _________:
        #   (b)

        h = up

    else:

        h = down

    _________
    #  (c)

    return _________(n, _________)
    #         (d)          (e)

def up(n, f):

    return 1 + f(3 * n + 1)

def down(n, f):
```

```
    return 1 + f(n // 2)
```

i. **(2 pt)** Fill in blank (a)?

lambda x, y : 1

ii. **(2 pt)** Fill in blank (b)?

n%2 != 0

iii. **(2 pt)** Fill in blank (c)?

g(n)

iv. **(1 pt)** Which of these could fill in blank (d)?

- ○ `f`
- ○ `g`
- ○ `hailstone`
- ○ `up`
- ○ `down`
- ● `h`

v. **(2 pt)** Fill in blank (e)?

lambda x :hailstone (x, g)

(b) (7 points)

Implement `collide`, which takes positive integers `m` and `n`. It returns the earliest element of the hailstone sequence starting at `n` that also appears in the hailstone sequence starting at `m`. Assume `hailstone` is implemented correctly.

```
def collide(m, n):
    """Return the earliest number in the hailstone sequence starting at n that
    also appears in the hailstone sequence starting at m.

    >>> collide(10, 32)  # 10, 5, 16, 8, ...  vs  32, 16, 8, ...
    16
    >>> collide(13, 11)  # 13, 40, ...  vs  11, 34, 17, 52, 26, 13, 40, ...
    13
    """

    s = []

    hailstone(m, _________)
    #               (a)

    found = None

    def f(k):

        _________
        #  (b)

        if _________:
        #     (c)

            _________
            #  (d)

    hailstone(n, f)

    return _________
    #         (e)
```

i. (2 pt) Fill in blank (a)?

s.append

ii. (1 pt) Fill in blank (b)?

nonlocal found

iii. (2 pt) Fill in blank (c)?

if k in s and found is None

iv. **(1 pt)** Which of the these could fill in blank (d)?

- ○ `return min(k, n)`
- ○ `found.append(min(k, n))`
- ● `found = k`
- ○ `return k`
- ○ `found.append(k)`
- ○ `found = min(k, n)`

v. **(1 pt)** Which of the these could fill in blank (e)?

- ○ `f(n)`
- ○ `s[-1]`
- ● `found`
- ○ `found[-1]`
- ○ `f(m)`
- ○ `s[0]`
- ○ `found[0]`
- ○ `s`

3. (21 points) College Party

In a US presidential election, each state has a number of electors.

Definition: For some collection of states `s`, a *win by at least* ***k*** is a (possibly empty) subset `w` of `s` such that the total number of electors for the states in `w` is at least `k` more than the total number of electors for the states not in `w` but in `s`.

For example, in the `battleground` states below, Arizona (`AZ`), Pennsylvania (`PA`), and Michigan (`MI`) have a total of 11 + 20 + 16 = 47 electors. The remaining states have a total of 6 + 16 + 10 = 32 electors. So, the subset `<AZ PA MI>` is a win by 47 - 32 = 15.

```
class State:
    electors = {}
    def __init__(self, code, electors):
        self.code = code
        self.electors = electors
        State.electors[code] = electors

battleground = [State('AZ', 11), State('PA', 20), State('NV', 6),
                State('GA', 16), State('WI', 10), State('MI', 16)]
```

The total number of electors for an empty set of states is 0.

The `print_all` function prints all elements of an iterable.

```
def print_all(s):
    for x in s:
        print(x)
```

(a) (8 points)

Implement `wins`, a generator function that takes a list of `State` instances `states` and an integer `k`. For every possible *win by at least* ***k*** among the `states`, it yields a **linked list containing strings** of the two-letter codes for the states in that win.

Any order of the wins and any order of the states within a win is acceptable.

A linked list is a `Link` instance or `Link.empty`. The `Link` class appears on the Midterm 2 Study Guide.

```
def wins(states, k):
    """Yield each linked list of two-letter state codes that describes a win by at least k.

    >>> print_all(wins(battleground, 50))
    <AZ PA NV GA WI MI>
    <AZ PA NV GA MI>
    <AZ PA GA WI MI>
    <PA NV GA WI MI>
    >>> print_all(wins(battleground, 75))
    <AZ PA NV GA WI MI>
    """

    if _________:
        # (a)

        yield Link.empty

    if states:

        first = states[0].electors
```

```
for win in wins(states[1:], _________):
    #                           (b)

    yield Link(_________, win)
    #             (c)

yield from wins(states[1:], _________)
#                               (d)
```

i. **(2 pt)** Which of the these could fill in blank (a)?

- ○ `k <= 0`
- ○ `k >= 0 and not states`
- ○ `not states`
- ● `k <= 0 and not states`
- ○ `k == 0`
- ○ `k == 0 and not states`
- ○ `k >= 0`

ii. **(2 pt)** Which of the these could fill in blank (b)?

- ● `k - first`
- ○ `first`
- ○ `k`
- ○ `k + first`
- ○ `max(k, first)`
- ○ `min(k, first)`
- ○ `0`
- ○ `-k`

iii. **(2 pt)** Fill in blank (c).

states[0].code

iv. **(2 pt)** Which of the these could fill in blank (d)?

- ● `k` ✗
- ○ `first`
- ○ `k - first`
- ○ `min(k, first)`
- ○ `-k`
- ○ `0`
- ○ `k + first` ✓
- ○ `max(k, first)`

(b) (7 points)

Implement `must_win`, which takes a list of `State` instances `states` and an integer `k`. It returns a list of two-letter state codes (strings) for all states that appear in every *win by at least* `k` among the `states`. Assume `wins` is implemented correctly.

```
def must_win(states, k):
    """List all states that must be won in every scenario that wins by k.

    >>> must_win(battleground, 50)
    ['PA', 'GA', 'MI']
    >>> must_win(battleground, 75)
    ['AZ', 'PA', 'NV', 'GA', 'WI', 'MI']
    """
    def contains(s, x):
        """Return whether x is a value in linked list s."""

        return (_________) and (_________)
        #          (a)             (b)

    return [_________ for s in states if _________([_________ for w in wins(states, k)])]
    #          (c)                          (d)        (e)
```

i. (1 pt) Which of these could fill in blank (a)?

- ○ `x == s.first`
- ● `s is not Link.empty`
- ○ `x not in s`
- ○ `x in s`
- ○ `s is Link.empty`
- ○ `x != s.first`

ii. (2 pt) Fill in blank (b).

True if x == s.first else contains(s.rest, x)

iii. (1 pt) Fill in blank (c).

s.code

iv. (1 pt) Fill in blank (d) with a single function name.

any ✗ (all)

v. (2 pt) Fill in blank (e).

contains(w, s.code)

(c) (6 points)

Definition. A win by at least `k` is *minimal* if every state in it is necessary to win by at least `k`.

The `State` class and `battleground` list are repeated here for convenience. Assume that only this code has been executed. You may not call `wins` or `must_win`.

```
class State:
    electors = {}
    def __init__(self, code, electors):
        self.code = code
        self.electors = electors
        State.electors[code] = electors

battleground = [State('AZ', 11), State('PA', 20), State('NV', 6),
                State('GA', 16), State('WI', 10), State('MI', 16)]
```

Implement `is_minimal`, which takes a non-empty list of strings `state_codes` in which every element is the code for some `State` instance, as well as an integer `k`. It returns whether the states named in `state_codes` form a minimal win by `k` among all `State` instances that have ever been constructed.

```
def is_minimal(state_codes, k):
    """Return whether a non-empty list of state_codes describes a minimal win by
    at least k.

    >>> is_minimal(['AZ', 'NV', 'GA', 'WI'], 0)  # Every state is necessary
    True
    >>> is_minimal(['AZ', 'GA', 'WI'], 0)        # Not a win
    False
    >>> is_minimal(['AZ', 'NV', 'PA', 'WI'], 0)  # NV is not necessary
    False
    >>> is_minimal(['AZ', 'PA', 'WI'], 0)        # Every state is necessary
    True
    """
    assert state_codes, 'state_codes must not be empty'

    votes_in_favor = _________
    #                   (a)

    total_possible_votes = sum(_________)
    #                              (b)

    def win_margin(n):
        "Margin of victory if n votes are in favor and the rest are against."

        return n - (total_possible_votes - n)

    if win_margin(sum(votes_in_favor)) < k:

        return False  # Not a win

    in_favor_no_smallest = _________
    #                          (c)

    return win_margin(in_favor_no_smallest) < k
```

i. **(2 pt)** Fill in blank (a). **You may not write `battleground` in your response.**

```
[states.electors[code] for code in state_codes]
```

ii. **(2 pt)** Fill in blank (b). **You may not write `battleground` in your response.**

```
[y for x , y in state.elector.items]
```

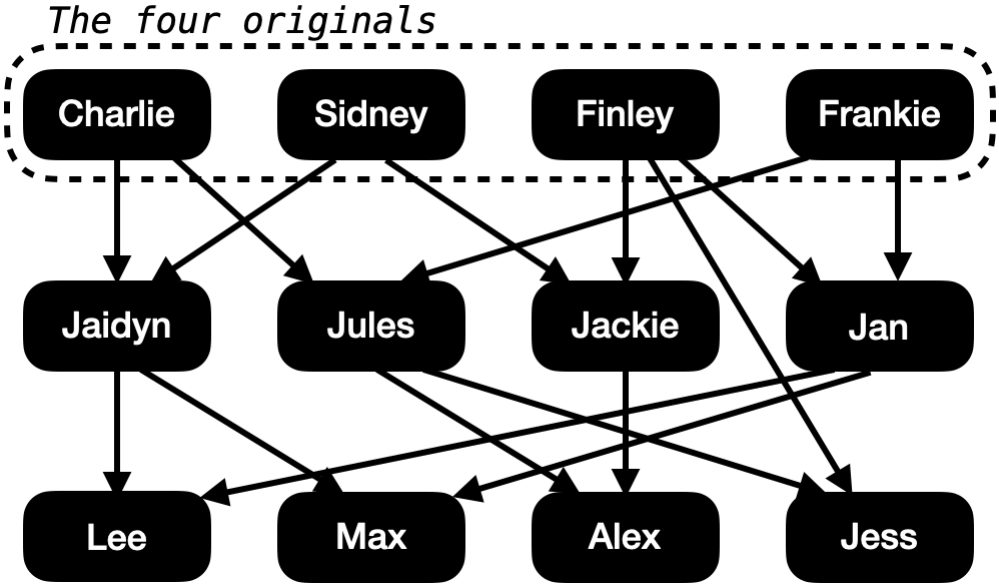
iii. **(2 pt)** Which of these could fill in blank (c). Check all that apply.

Hint: Two states may have the same number of electors.

- [x] `sum(votes_in_favor - min(votes_in_favor))`
- [ ] `sum(votes_in_favor) - min(votes_in_favor)`
- [ ] `sum(votes_in_favor - [min(votes_in_favor)])`
- [ ] `sum(votes_in_favor.remove(min(votes_in_favor)))`
- [ ] `sum([v for v in votes_in_favor if v > min(votes_in_favor)]`

4. (22 points) Last Lecture AMA

Llambda the llama breeder had four *original* llamas, but now has 12. An arrow from one llama to another indicates that the first is a parent of the second. For example, Jackie's parents are Sidney and Finley.

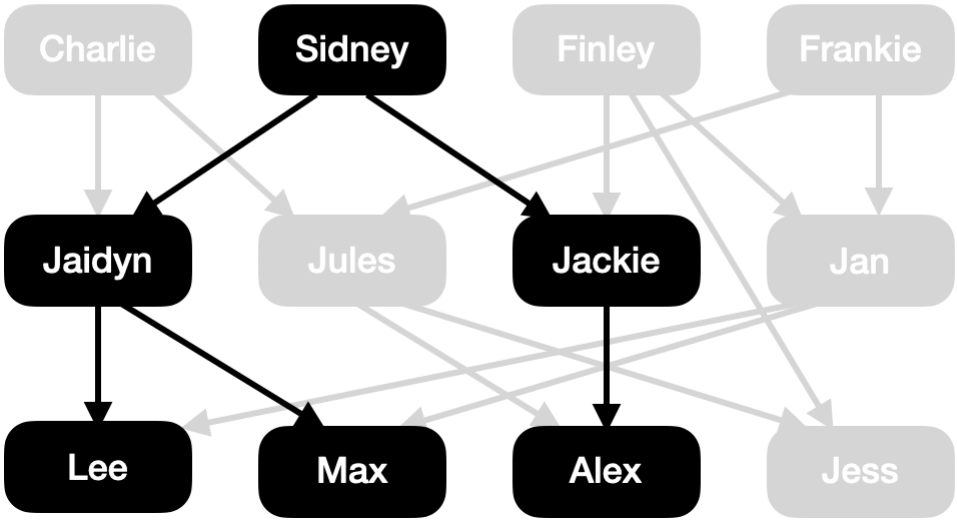


All llamas except the originals have 2 parents, and each has a unique name.

Definition. An *offspring tree* is a `Tree` instance with string labels in which each node represents a llama and the branches of a node represent its (biological) children.

Sidney's offspring tree:

```
Tree('Sidney', [Tree('Jaidyn', [Tree('Lee'),
                                Tree('Max')]),
                Tree('Jackie', [Tree('Alex')])])
```



The `Tree` class appears on the Midterm 2 Study Guide.

Assume `originals` is a list of offspring trees for the original four.

```
originals = [Tree('Charlie', ...), Tree('Sidney', ...),
```

```
        Tree('Finley', ...), Tree('Frankie', ...)]
```

(a) (6 points)

Implement `related`, which takes two strings `a` and `b` that are names, as well as a list of `offspring_trees` for the originals. It returns whether `a` and `b` are related. That is, they either share a common ancestor or one is an ancestor of the other.

```
def related(a, b, offspring_trees):
    """Return whether the llamas named a and b are related.

    >>> related('Charlie', 'Max', originals)   # Grandparent
    True
    >>> related('Jules', 'Jackie', originals)  # Not related, even though they have child
    False
    >>> related('Max', 'Jules', originals)     # Both descend from Charlie and Frankie
    True
    >>> related('Max', 'Jess', originals)      # Both descend from Charlie and Finley
    True
    """

    def brood(t):
        """Return a list of the names of all llamas in Tree t."""

        result = _________
        #           (a)

        for b in t.branches:

            result._________(_________)
            #         (b)       (c)

        return result

    for s in _________:
        #       (d)

        if a in s and b in s:

            return True

    return False
```

i. (1 pt) Which of these could fill in blank (a)?

- ○ `list(t.branches)`
- ○ `[t]`
- ○ `[]`
- ○ `[b.label for b in t.branches]`
- ● `[t.label]`

ii. **(1 pt)** Which of these could fill in blank (b)?

- ○ `insert`
- ● `extend`
- ○ `append`
- ○ `pop`
- ○ `remove`

iii. **(2 pt)** Fill in blank (c).

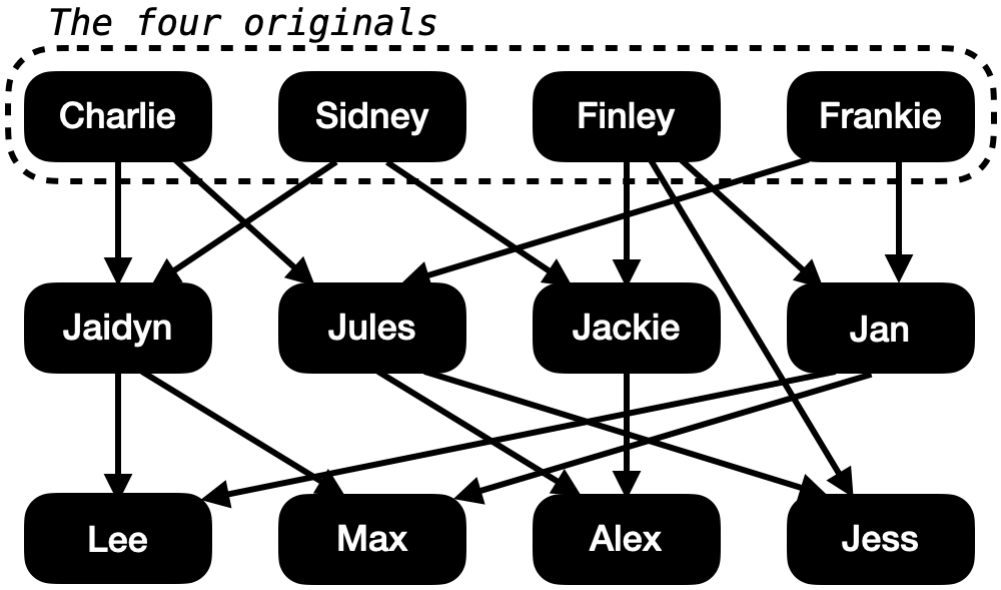
brood(b)

iv. **(2 pt)** Fill in blank (d).

[brood(t) for t in offspring_tress]

(b) (8 points)

This figure is repeated for convenience:



Implement `parents`, which takes two strings `a` and `b` that are names of llamas, as well as a list of `offspring_trees` for the originals. It returns whether `a` and `b` are both parents of the same child.

```
def parents(a, b, offspring_trees):
    """Return whether a and b are both parents of the same child.

    >>> parents('Jules', 'Jackie', originals)  # Parents of Alex
    True
    >>> parents('Jules', 'Finley', originals)  # Parents of Jess
    True
    >>> parents('Jules', 'Jaidyn', originals)
    False
    >>> parents('Jules', 'Sidney', originals)
    False
    """

    data = {}

    def traverse(t):

        for b in _________:
            #       (a)

            if _________:
                # (b)

                data[b.label] = []

            data[b.label].__________
            #               (c)


            _________
            #  (d)
```

```
for t in _________:
    #       (e)

    traverse(t)

return _________([a in s and b in s for s in data._________])
#         (f)                                        (g)
```

i. **(1 pt)** Which of these could fill in blank (a)?

- ○ `branches(b)`
- ○ `b.branches`
- ○ `branches(t)`
- ● `t.branches`

ii. **(1 pt)** Fill in blank (b).

b.label not in data

iii. **(2 pt)** Fill in blank (c).

append(t.label)

iv. **(1 pt)** Fill in blank (d).

traverse(b)

v. **(1 pt)** Which of these could fill in blank (e)?

- ○ `map(list, offspring_trees)`
- ○ `filter(list, offspring_trees)`
- ○ `map(traverse, offspring_trees)`
- ● `offspring_trees`
- ○ `filter(traverse, offspring_trees)`

vi. **(1 pt)** Fill in blank (f) with a single function name.

any

vii. (1 pt) Which of these could fill in blank (g)?

- ○ `items()`
- ○ `keys()`
- ○ `copy()`
- ● `values()`

(c) (4 points)

The llamas are described by a two-column SQL table named `brood` that has one row for each parent-child pair.

```
parent  | child
========|========
Charlie | Jaidyn
Sidney  | Jaidyn
Charlie | Jules
Jules   | Alex
Jules   | Jess
...     | ...
```

Select a two-column table with one row for each llama that has children. The first column is the name of the llama and the second column is the number of children it has.

```
SELECT _________;
         (a)
```

i. (4 pt) Fill in blank (a).

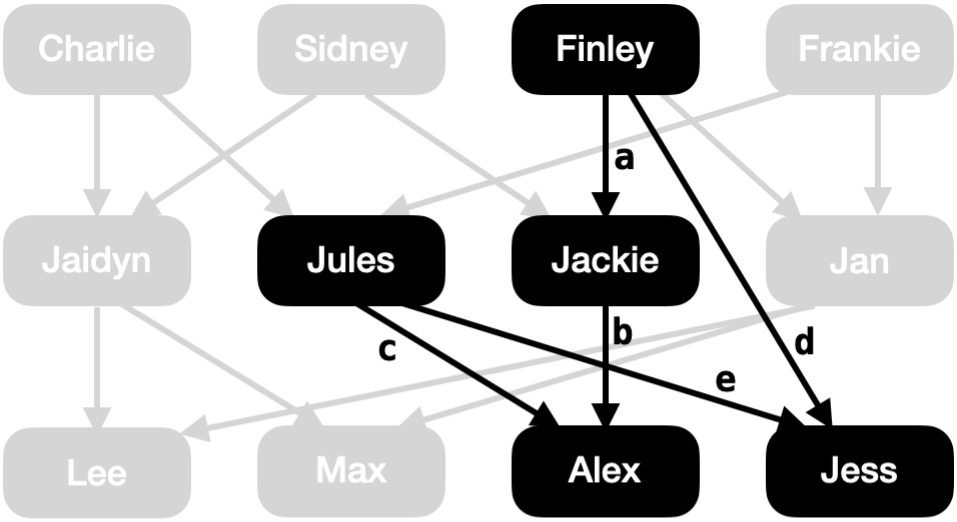
SELECT parent, COUNT(child) FROM brood GROUP BY parent

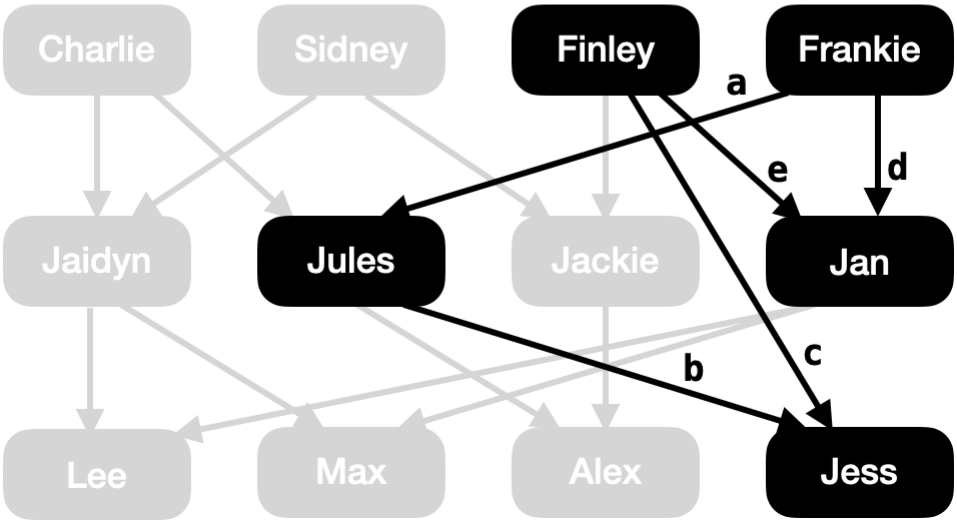
(d) (6 points)

Definition. A llama `p` is *multigenerational* if it has a child whose other parent is also a parent of `p`'s grandchild.

For example:

- Finley has a child Jess whose other parent Jules is also a parent of Alex, which is Finley's grandchild.
- Frankie has a child Jan whose other parent Finley is also a parent of Jess, which is Frankie's grandchild.





Hint: In the diagram above, the labels **a** through **e** correspond to the five joined tables in the query and are meant to help you.

Complete a query that selects a one-column table with the name of each *multigenerational* llama. The result has two rows: `Finley` and `Frankie`.

```
SELECT a.parent FROM brood AS a, brood AS b, brood AS c, brood AS d, brood AS e

    WHERE _____ AND _____ AND _____ AND _____ AND _____ AND _____;
          (u)       (v)       (w)       (x)       (y)       (z)
```

i. **(1 pt)** Which of these could fill in blank (u)?

- [x] `a.parent = b.parent`
- [ ] `a.child = b.parent`
- [ ] `a.child = b.child`

ii. **(1 pt)** Which of these could fill in blank (v)?

- [ ] `b.parent = c.child`
- [ ] `b.parent != c.child`
- [ ] `b.parent = c.parent`
- [x] `b.parent != c.parent`

iii. **(1 pt)** Which of these could fill in blank (w)?

- [ ] `b.child < c.child`
- [ ] `b.child != c.child`
- [x] `b.child = c.child`

iv. **(1 pt)** Which of these could fill in blank (x)?

- [x] `a.parent != d.parent`
- [ ] `a.parent = d.parent`
- [ ] `a.parent != d.child`
- [ ] `a.parent = d.child`

v. **(1 pt)** Which of these could fill in blank (y)?

- [x] `c.parent = e.parent`
- [ ] `c.parent != e.parent`
- [ ] `c.parent != e.child`
- [ ] `c.parent = e.child`

vi. **(1 pt)** Which of these could fill in blank (z)?

- [ ] `d.child = e.child`
- [ ] `d.child != e.parent`
- [x] `d.child = e.parent`
- [ ] `d.child != e.child`

5. (10 points) SchemeQL

(a) (4 points)

Implement the procedure `cons` that behaves just like the built-in `cons` when called on a value `x` and a (possibly empty) list `s`. You may not write `cons` in your solution.

```
(define (cons x s) ( _________ _________ _________ ))
;                        (a)       (b)       (c)
```

i. (1 pt) Which of these could fill in blank (a)?

- (●) `append`
- ( ) `map`
- ( ) `lambda`
- ( ) `list`
- ( ) `if`

ii. (2 pt) Fill in blank (b).

(list x)

iii. (1 pt) Which of these could fill in blank (c)? Check all that apply.

- [■] `s`
- [ ] `(car s)`
- [ ] `(list s)`
- [ ] `(cdr s)`

(b) (6 points)

The `join` procedure takes two lists of lists `s` and `t`. It returns a list of lists that has one element for each possible pairing of an element of `s` with an element of `t`. Each element of the result is a list that has all the elements of a list from `s` followed by all the elements of a list from `t`.

For example:

```
scm> (define instructors '(
         (john 61a)
         (hany 61a)
         (josh 61b)))
instructors
scm> (define grades '(
         (a b)
         (c d)))
grades
scm> (join instructors grades)
((john 61a a b) (john 61a c d) (hany 61a a b) (hany 61a c d) (josh 61b a b) (josh 61b c d))
```

Implement `join`.

```
(define (join s t)

  (if (null? s) nil

      (_________ (_________ (lambda (v) (_________ _________ _________)) t)
      ;   (a)        (b)                    (c)       (d)       (e)

              (join _________ t))))
              ;        (f)
```

i. (1 pt) Fill in blank (a) with a single procedure name or symbol.

cons ✗ append

ii. (1 pt) Fill in blank (b) with a single procedure name or symbol.

map

iii. (1 pt) Fill in blank (c) with a single procedure name or symbol.

append

iv. (1 pt) Fill in blank (d).

(car s)

v. **(1 pt)** Which of these could fill in blank (e)?

- (●) `v`
- ( ) `t`
- ( ) `s`

vi. **(1 pt)** Fill in blank (f).

No more questions.