CS 61A
Spring 2021

Structure and Interpretation of Computer Programs

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?

1. (6.0 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
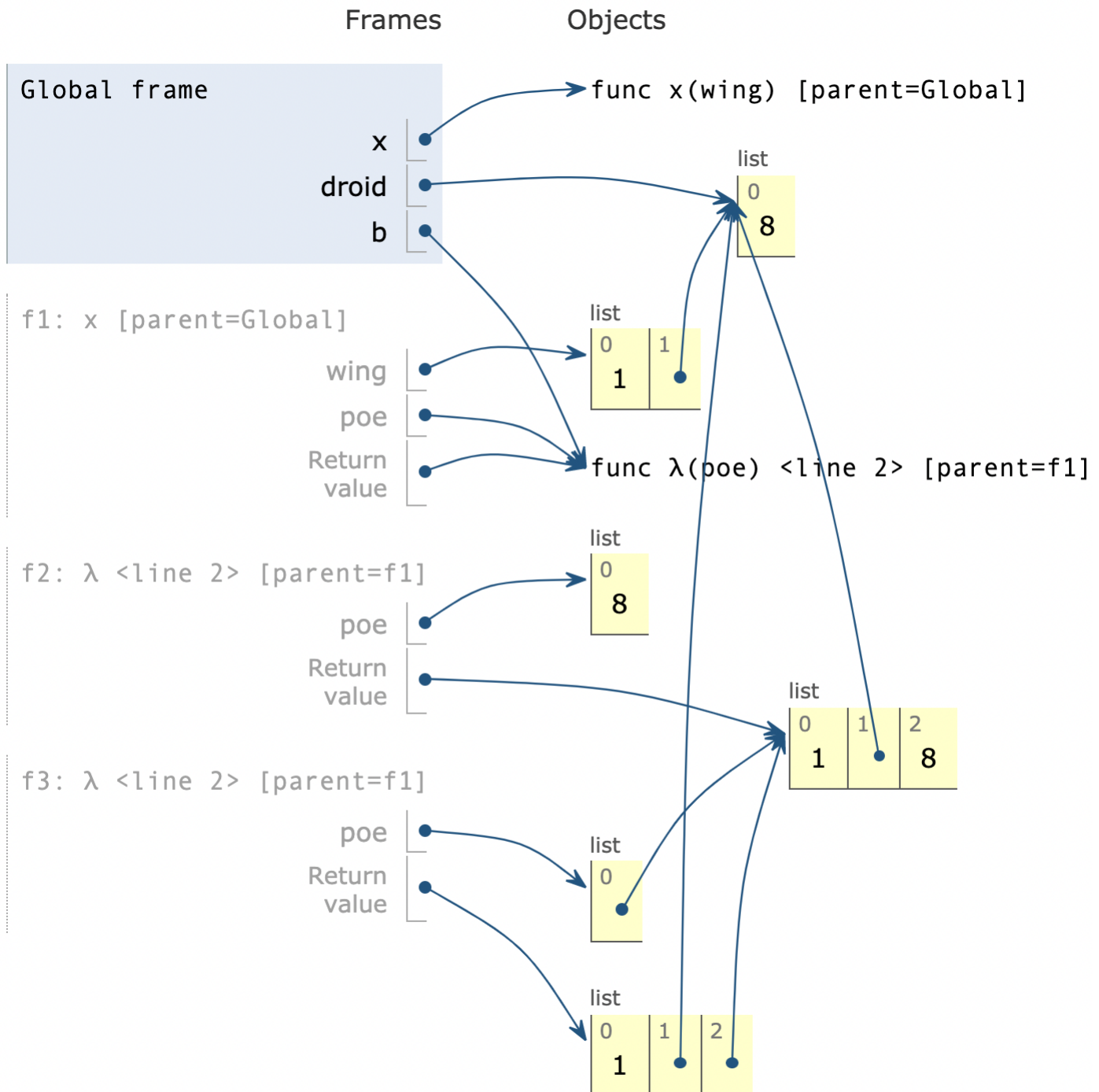

```
def x(wing):

    poe = lambda poe: _____
```

```
    #                    (a)

    wing.append(_____)
    #               (b)

    return _____
    #          (c)

droid = [8]

b = x([1])

_____
#  (d)
```

(a) (1.0 pt) Which of these could fill in blank (a)? Check all that apply.

☑ `wing + poe`

☐ `wing.extend(poe)`

☐ `wing.append(poe)`

☐ `list(wing).extend(poe)`

☐ `list(wing).append(poe)`

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

> **droid**

(c) (1.0 pt) Which of these could fill in blank (c)?

● `poe`

○ `poe(droid)`

○ `poe(wing)`

○ `poe(b)`

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

> **b([b([8])])**

2. (8.0 points)   It's Raining Stones!

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) 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]
    """

def up(n, f):

    return 1 + f(3 * n + 1)

def down(n, f):

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

i. Here is a skeleton of a correct solution:

```
def hailstone(n, g):
    if n == 1:
        h = _____
    elif _____:
        h = up
    else:
        h = down
    _____
    return _____(n, _____)
```

We highly encourage you to follow the skeleton. You may diverge from the skeleton if your approach is correct and passes the doctests, but a non-working solution that doesn't follow the skeleton will not receive partial credit.

```
def hailstone(n, g):
    if n == 1:
        h = lambda n, f: 1
    elif n % 2 == 1:
        h = up
    else:
        h = down
    g(n)
    return h(n, lambda n: hailstone(n, g))
```

3. **(8.0 points)   Compression**

(a) **Definition:** A compression is when two adjacent elements in the list are either added or subtracted from each other to form one single new element.

For example, for the list `[1,2,1]` the first compression could result in either: `[3, 1]` (addition) or `[-1, 1]` (subtraction)

Write a function `compress` that takes in two lists of positive integers, `lst1` and `lst2`, and determines if `lst1` can be compressed into `lst2`.

```
def compress(lst1, lst2):
    """Takes in two lists of positive integers, lst1 and lst2, and returns True
    if lst1 can be compressed into lst2, and False otherwise

    >>>> compress([1,1,1], [3])
    True
    >>>> compress([1,1,1], [2])
    False
    >>>> compress([1,1,1], [1])
    True
    >>>> compress([1,2,3], [1,5])
    True
    >>>> compress([1,2,3], [2])
    True
    >>>> compress([], [1,2,3])
    False
    >>>> compress([1,2,3], [])
    False
    >>>> compress([], [])
    True
    >>>> compress([1,4,2,8,3,9,4], [31])
    True
    >>>> compress([1,4,2,8,3,9,4], [3,5,5])
    True
    """
```

i. Here is a skeleton of a correct solution:

```
def compress(lst1, lst2):
    if _____ == _____:
        return _____
    elif _____ or _____:
        return _____
    compress_add = _____
    compress_sub = _____
    compress_eq = _____ and _____
    return _____
```

We highly encourage you to follow the skeleton. You may diverge from the skeleton if your approach is correct and passes the doctests, but a non-working solution that doesn't follow the skeleton will not receive partial credit.

```
def compress(lst1, lst2):
    if lst1 == lst2:
        return True
    elif len(lst1) <= 1 or len(lst2) == 0:
        return False
    compress_add = compress([lst1[0] + lst1[1]] + lst1[2:], lst2)
    compress_sub = compress([lst1[0] - lst1[1]] + lst1[2:], lst2)
    compress_eq = compress(lst1[1:], lst2[1:]) and lst1[0] == lst2[0]
    return compress_eq or compress_add or compress_sub
```

4. (21.0 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) (8.0 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. (2.0 pt) Which of these could fill in blank (a)?

○ k >= 0

○ k <= 0

○ k == 0

○ not states

○ k >= 0 and not states

● k <= 0 and not states

○ k == 0 and not states

ii. (2.0 pt) Which of these could fill in blank (b)?

● k - first

○ k + first

○ k

○ -k

○ 0

○ first

○ min(k, first)

○ max(k, first)

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

states[0].code

iv. (2.0 pt) Which of these could fill in blank (d)?

○ k - first

● k + first

○ k

○ -k

○ 0

○ first

○ min(k, first)

○ max(k, first)

(b) **(7.0 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. **(1.0 pt)** Which of these could fill in blank (a)?

○ `s is Link.empty`

● `s is not Link.empty`

○ `x in s`

○ `x not in s`

○ `x == s.first`

○ `x != s.first`

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

> **s.first == x or contains(s.rest, x)**

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

> **s.code**

iv. **(1.0 pt)** Fill in blank (d) with a single function name.

> **all**

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

> **contains(w, s.code)**

(c) (6.0 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. **(2.0 pt)** Fill in blank (a). **You may not write** `battleground` **in your response.**

> **[State.electors[s] for s in state_codes]**

ii. **(2.0 pt)** Fill in blank (b). **You may not write** `battleground` **in your response.**

> **State.electors.values()**

iii. **(2.0 pt)** Which of these could fill in blank (c). Check all that apply.

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

- ☑ `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)])`

5. **(17.0 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.
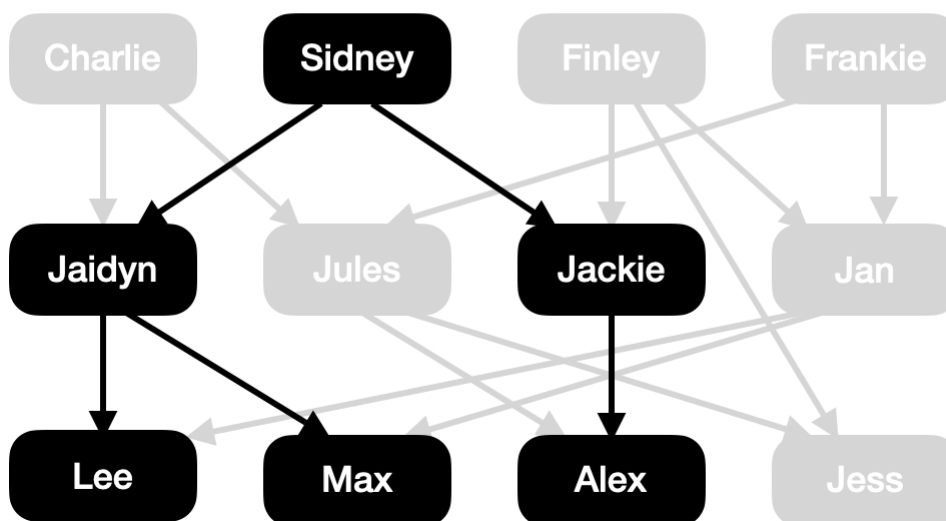

The four originals
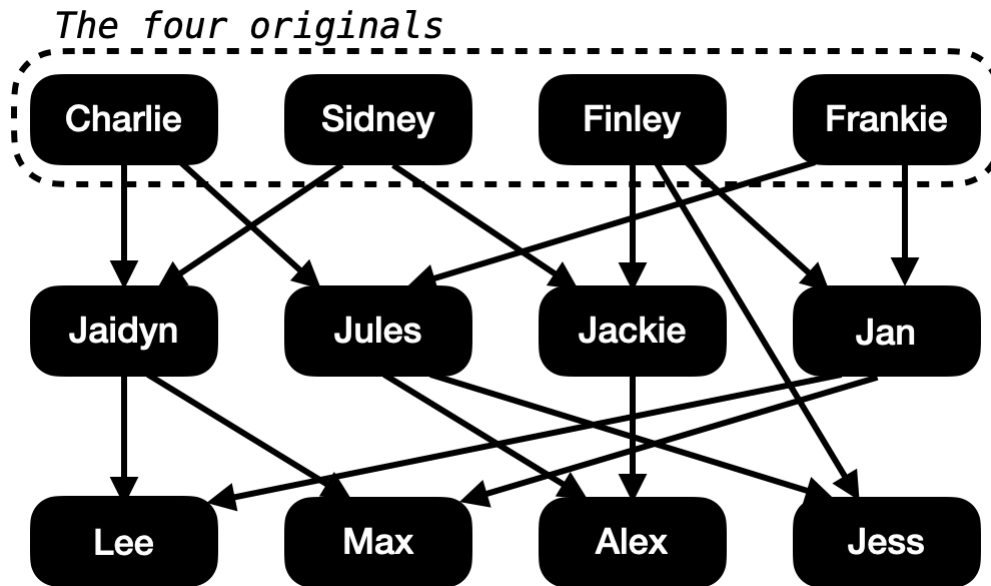
All llamas except the originals have 2 parents, and each has a unique name.

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) (7.0 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.

```python
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 family(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. (1.0 pt) Which of these could fill in blank (a)?

○ []

○ [t]

● [t.label]

○ list(t.branches)

○ [b.label for b in t.branches]

ii. **(1.0 pt)** Which of these could fill in blank (b)?

○ `append`

● `extend`

○ `pop`

○ `remove`

○ `insert`

iii. **(2.0 pt)** Fill in blank (c).
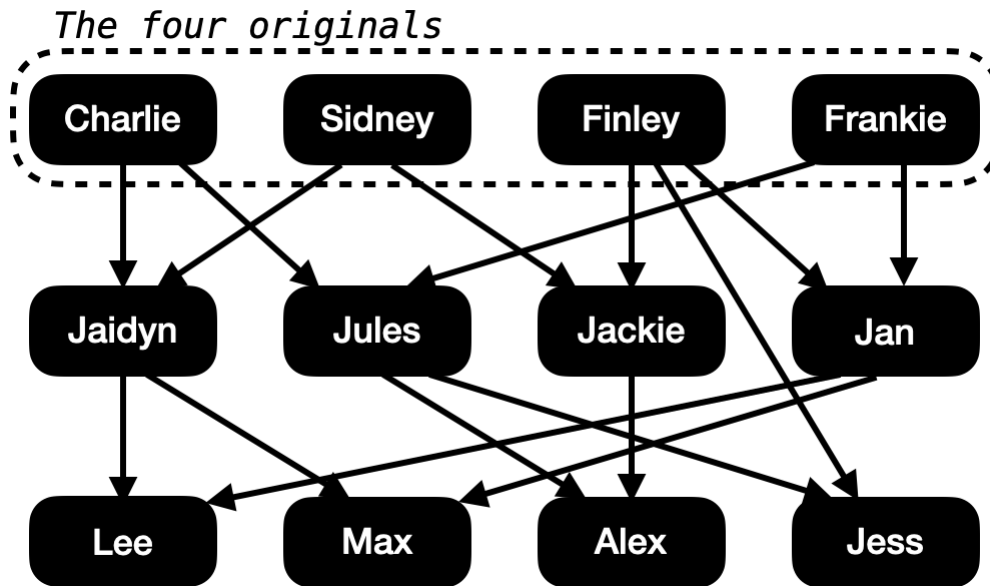
> **family(b)**

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

> **map(family, offspring_trees)**

v. **(1.0 pt)** What is the order of growth of `family(t)` in terms of the number of nodes `n` in Tree `t`?

○ (1) - constant

○ $(\log_2 n)$ - logarithmic

● (n) - linear

○ (n^2) - quadratic

○ (2^n) - exponential

○ None of These

(b) (8.0 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.

```python
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
    """

    storage = {}

    def traverse(t):

        for b in _____:
            #        (a)

            if _____:
                # (b)

                storage[b.label] = []

            storage[b.label]._____
            #                    (c)


            _____
            #   (d)
```

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

        traverse(t)

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

i. (1.0 pt) Which of these could fill in blank (a)?

- 🔵 `t.branches`
- ⚪ `b.branches`
- ⚪ `branches(t)`
- ⚪ `branches(b)`

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

> **b.label not in storage**

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

> **append(t.label)**

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

> **traverse(b)**

v. (1.0 pt) Which of these could fill in blank (e)?

- 🔵 `offspring_trees`
- ⚪ `map(traverse, offspring_trees)`
- ⚪ `map(list, offspring_trees)`
- ⚪ `filter(traverse, offspring_trees)`
- ⚪ `filter(list, offspring_trees)`

vi. (1.0 pt) Fill in blank (f) with a single function name.

> **any**

vii. (1.0 pt) Which of these could fill in blank (g)?

- 🔵 `values()`
- ⚪ `keys()`
- ⚪ `items()`
- ⚪ `copy()`

6. **(10.0 points)**   **SchemeQL**

  (a) **(4.0 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. **(1.0 pt)** Which of these could fill in blank (a)?

       ○ `list`

       ● `append`

       ○ `if`

       ○ `map`

       ○ `lambda`

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

> (list x)

    iii. **(1.0 pt)** Which of these could fill in blank (c)? Check all that apply.

       ■ `s`

       ☐ `(cdr s)`

       ☐ `(car s)`

       ☐ `(list s)`

(b) **(6.0 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. **(1.0 pt)** Fill in blank (a) with a single procedure name or symbol.

append

ii. **(1.0 pt)** Fill in blank (b) with a single procedure name or symbol.

map

iii. **(1.0 pt)** Fill in blank (c) with a single procedure name or symbol.

append

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

(car s)

v. **(1.0 pt)** Which of these could fill in blank (e)?

○ s

○ t

● v

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

(cdr s)

7. (2.0 points)   Don't Be Late For Class!

All of the EECS class description pages list the days and times for the class in a particular format. For example:

`MoWe 10:00AM - 10:59AM`

Write a regular expression that matches that date/time format and *does* not match similar formats. Consult the doctests for examples of what your regular expression should and should not match. All possible combinations of days should be matched.

Remember that you can use regexr.com to try out regular expressions and use code.cs61a.org to run the doctests.

Note below that the second and third blanks will be filled by the same expression (b).

```
import re

def find_time(class_descrip):
    """
    >>> find_time("(Fall 2021): MoWe 10:00AM - 10:59AM, Online - Hug")
    'MoWe 10:00AM - 10:59AM'
    >>> find_time("(Fall 2021): MoWeFr 10:00AM - 10:59AM, Lewis 100 - Nicholas Weaver")
    'MoWeFr 10:00AM - 10:59AM'
    >>> find_time("(Spring 2021): TuTh 12:30PM - 1:59PM, Online - Rao, Parekh")
    'TuTh 12:30PM - 1:59PM'
    >>> find_time("The class that never happens! TuTh 26:62PM - 1:62PM, The QuantumVerse")
    >>> find_time("Meet me on the glade from 2:00PM - 2:59PM today for a mochi muffin!")
    >>> find_time("My calendar is free TuTh 11-12 PM, how about yours?")
    >>> find_time("I wanna find a class for Fall 2021 that's @ Tuesday 9:00, seen any?")
    """
    match = re.search(r"_____\s_____ - _____", class_descrip)
    #                     (a)         (b)          (b)
    return match and match.group(0)
```

Here are the test cases in one block for your convenience:

```
(Fall 2021): MoWe 10:00AM - 10:59AM, Online - Hug
(Fall 2021): MoWeFr 10:00AM - 10:59AM, Lewis 100 - Nicholas Weaver
(Spring 2021): TuTh 12:30PM - 1:59PM, Online - Rao, Parekh
The class that never happens! TuTh 26:62PM - 1:62PM, The QuantumVerse
Meet me on the glade from 2:00PM - 2:59PM today for a mochi muffin!
My calendar is free TuTh 11-12 PM, how about yours?
I wanna find a class for Fall 2021 that's @ Tuesday 9:00, seen any?
```

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

((Mo|Tu|We|Th|Fr)+)

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

((2[0-3])|(1?\d)):[0-5]\d(AM|PM)

8. **(3.0 points)   Comprehension is Everything**

The following EBNF grammar can describe a subset of Python list comprehensions, but cannot yet describe all of them.

Remember that you can use code.cs61a.org to try out the grammar and parse expressions with it.

```
start: comp

?comp: "[" expression "for" IDENTIFIER "in" IDENTIFIER "]"

expression: IDENTIFIER operation*

operation: OPERATOR NUMBER

IDENTIFIER: /[a-zA-Z]+/

OPERATOR: "*" | "/" | "+" | "-"

%import common.NUMBER
%ignore /\s+/
```

(a) **(1.0 pt)** Select all of the non-terminal symbols in the grammar:

- ■ comp
- ■ expression
- ■ operation
- ☐ NUMBER
- ☐ IDENTIFIER
- ☐ OPERATOR

(b) **(1.0 pt)** Which of the following comprehensions would be successfully parsed by the grammar?

- ■ [x * 2 for x in list]
- ■ [x for x in list]
- ☐ [x ** 2 for x in list]
- ☐ [x + 2 for x in list if x == 1]
- ☐ [x * y for x in list for y in list2]
- ☐ [x - 2 for x in my_list]
- ☐ [x - y for (x,y) in tuples]

(c) **(1.0 pt)** Which line would we need to modify to add support for a % operator, like in the expression [n % 2 for n in numbers]?

- ● OPERATOR: "*" | "/" | "+" | "-"
- ○ IDENTIFIER: /[a-zA-z]+/
- ○ operation: OPERATOR NUMBER
- ○ expression: IDENTIFIER operation*
- ○ ?comp: "[" expression "for" IDENTIFIER "in" IDENTIFIER "]"

No more questions.