CS 61A Fall 2024

Structure and Interpretation of Computer Programs

Final Solutions

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 cut(s):
    this = 1
    for x in s:
        if this:
            this = x
            yield this
        else:
            this = True

def paste(n):
    yield n
    for x in paste(n + 1):
        yield 2 * x
    yield n

def copy(t, k):
    return [next(t) for x in range(k)]

nums = [0, 2, 4, 6, 8]
```

Write the output that would be displayed by printing the result of each expression. If an error occurs, write ERROR.

(a) (2.0 pt) `list(cut(nums))`

- ○ `[0, 2, 4, 6, 8]`
- ● `[0, 4, 6, 8]`
- ○ `[1, 0, 2, 4, 6, 8]`
- ○ `[1, 0, 4, 6, 8]`
- ○ `[1, 2, 4, 6, 8]`
- ○ `[2, 4, 6, 8]`

(b) (2.0 pt) `list(cut(map(lambda x: x - 4, cut(nums))))`

```
[-4, 0, 4]
```

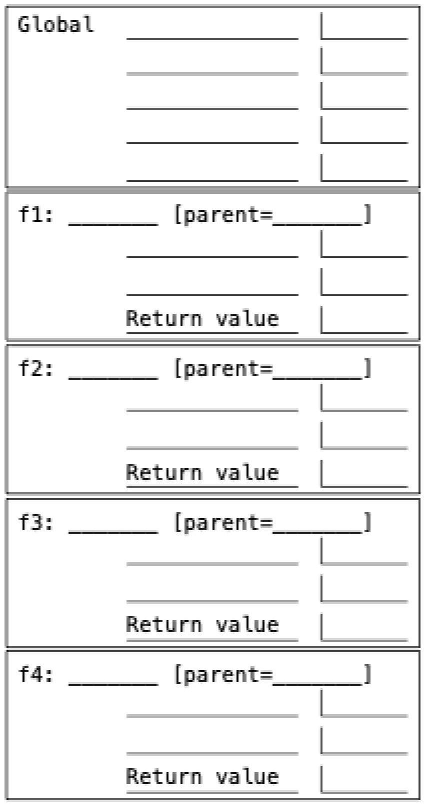
(c) (2.0 pt) `copy(paste(0), 3)`

```
[0, 2, 8]
```

2. (4.0 points) Line Diagram

Complete the environment diagram to answer the following questions. Only the questions will be scored.

```
 1: b = 6
 2: log = []
 3:
 4: def make_line(a):
 5:     b = 1
 6:     list(log).append(a)
 7:     return lambda x: a * x + b
 8:
 9: def shift_line(f):
10:     b = 3
11:     log.append([b])
12:     return lambda y: f(y) + b
13:
14: z = shift_line(make_line(2))(0)
15: print(log)
16: print(z)
```



(a) (2.0 pt) What is displayed by `print(log)` on line 15?

- ○ `[]`
- ○ `[3]`
- ● `[[3]]`
- ○ `[[[3]]]`
- ○ `[2, 3]`
- ○ `[2, [3]]`
- ○ `[2, [[3]]]`
- ○ `[[2], [3]]`
- ○ `[[2], [[3]]]`

(b) (2.0 pt) What is displayed by `print(z)` on line 16?

- ○ 2
- ● 4
- ○ 6
- ○ 7
- ○ 9
- ○ 12

3. (5.0 points) Make Tens

A list of numbers is easier for a person to sum up if it can be split into groups that all sum to 10. Implement `tens`, which takes a list of positive numbers `s`. It returns `True` if, for every positive integer `i` where `10 * i <= sum(s)`, there is a positive `k` that satisfies `sum(s[0:k]) == 10 * i`. Otherwise, it returns `False`.

```
def tens(s):
    """Return whether every multiple of 10 less than or equal to sum(s) appears as a prefix of s.
    >>> tens([3, 2, 2, 3, 6, 2, 2, 4, 1, 5, 2]) # sum(s[:4])==10, sum(s[:7])==20, sum(s[:10])==30
    True
    >>> tens([3, 2, 2, 3, 6, 2, 6, 1, 5, 2]) # sum(s[:4])==10, but no slice starting at 0 sums to 20
    False
    """
    t = 0
    for x in s:
        t += x
        if _______:
             (g)
            return _______
                     (h)
        if t == _______:
                   (i)
            t = _______
                  (j)
    return True
```

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

○ `t == 10`

○ `t != 10`

● `t > 10`

○ `t < 10`

(b) (2.0 pt) Fill in blank (h). **Select all correct answers.**

■ `False`

☐ `t % 10 == 0`

☐ `t % 10 != 0`

☐ `sum(s) % 10 == 0`

☐ `sum(s) % 10 != 0`

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

○ `0`

● `10`

○ `x`

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

0

4. (13.0 points) Count Misses

(a) (6.0 points)

Implement the `Counter` class. A `Counter` has a `count` of the number of times `inc` has been invoked on itself or any of its offspring. Its offspring are the Counters created by its `spawn` method or the `spawn` method of any of its offspring.

```
class Counter:
    """Counts how many times inc has been invoked on itself or any of its offspring.

    >>> total = Counter()
    >>> odd, even = total.spawn(), total.spawn()  # these are offspring of total
    >>> one, three = odd.spawn(), odd.spawn()     # these are offspring of odd and total
    >>> for c in [one, even, three, even, odd, even]:
    ...     c.inc()
    >>> [c.count for c in [one, three, even, odd, total]]
    [1, 1, 3, 3, 6]
    """
    def __init__(self, parent=None):
        self.parent = parent
        _______
          (a)

    def inc(self):
        self.count += 1
        _______:
          (b)
            _______
              (c)

    def spawn(self):
        return _______
                 (d)
```

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

`self.count = 0`

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

- ○ `if parent is not None:`
- ● `if self.parent is not None:`
- ○ `while parent is not None:`
- ○ `while self.parent is not None:`
- ○ `for p in parent:`
- ○ `for p in self.parent:`

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

- ○ `p += 1`
- ○ `p.count += 1`
- ○ `p.inc()`
- ○ `p.count.inc()`
- ○ `parent += 1`
- ○ `parent.count += 1`
- ○ `parent.inc()`
- ○ `parent.count.inc()`
- ○ `self.parent += 1`
- ○ `self.parent.count += 1`
- ● `self.parent.inc()`
- ○ `self.parent.count.inc()`

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

- ○ `self.parent`
- ○ `self.parent.spawn()`
- ○ `self.spawn()`
- ○ `Counter()`
- ○ `Counter().spawn()`
- ● `Counter(self)`
- ○ `Counter(self.count)`

(b) (7.0 points)

Implement the `MissDict` class. A `MissDict` has a dictionary `d`. Its `get` method takes an iterable `keys`, returns a list of all values in `d` that correspond to those `keys`, and counts the number of `keys` that did not appear in `d` (called *misses*). Printing a `MissDict` displays a fraction in which:

- The numerator is the number of misses during all calls to `get` for that particular `MissDict` instance.
- The denominator is the number of misses during all calls to `get` for any `MissDict` instance.

Assume `Counter` is implemented correctly.

```
class MissDict:
    """Has a dict, gets a list of values for an iterable of keys,
    and counts the number of keys that are not in the dict.

    >>> double = MissDict({1: 2, 2: 4, 3: 6, 5: 10})
    >>> half = MissDict({2: 1.0, 3: 1.5, 4: 2.0})
    >>> double.get([1, 3, 5, 2, 4])  # No value for key 4 (1 miss)
    [2, 6, 10, 4]
    >>> double.get([5, 4, 3, 0, 4])  # No value for keys 4 or 0 or 4 (3 misses)
    [10, 6]
    >>> half.get([1, 3, 5, 2, 4])    # No value for keys 1 or 5 (2 misses)
    [1.5, 1.0, 2.0]
    >>> print(double)  # double had 4 misses & half had 2 misses
    4/6 of the misses
    """
    misses = Counter()
    def __init__(self, d):
        assert isinstance(d, dict)
        self.d = d
        self.misses = _______
                        (e)
    def get(self, keys):
        result = []
        for k in keys:
            if k in self.d:
                _______
                  (f)
            else:
                _______
                  (g)
        return result

    def __str__(self):
        return f'_______ of the misses'
                   (h)
```

i. (2.0 pt) Fill in blank (e). **Select all correct answers.**

- ☐ `MissDict.spawn()`
- ■ `MissDict.misses.spawn()`
- ☐ `Counter()`
- ☐ `Counter(MissDict)`
- ■ `Counter(MissDict.misses)`

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

```
result.append(self.d[k])
```

iii. (1.0 pt) Fill in blank (g).

- ○ `misses.inc()`
- ○ `misses.count += 1`
- ● `self.misses.inc()`
- ○ `self.misses.count += 1`
- ○ `MissDict.misses.inc()`
- ○ `MissDict.misses.count += 1`

iv. (2.0 pt) Fill in blank (h).

- ○ `{misses.count / MissDict.misses.count}`
- ○ `{misses.count} / {MissDict.misses.count}`
- ○ `{misses.count / self.MissDict.misses.count}`
- ○ `{misses.count} / {self.MissDict.misses.count}`
- ○ `{self.misses.count / misses.count}`
- ○ `{self.misses.count} / {misses.count}`
- ○ `{self.misses.count / MissDict.misses.count}`
- ● `{self.misses.count} / {MissDict.misses.count}`
- ○ `{self.misses.count / self.MissDict.misses.count}`
- ○ `{self.misses.count} / {self.MissDict.misses.count}`

5. (16.0 points) Promotion to CS 61B

Definition. Given a sequence `s` and positions (indices) `i` and `j` in `s` with `i < j`, *promoting* an element to `i` from `j` means reordering `s` so that the element originally at position `j` is now at position `i`, all elements originally positioned between `i` and `j` increase their position (index) by one, and all other elements stay where they are. The beginning of `s` is position 0. For example, in the list `[30, 60, 90, 120, 150, 180]`, promoting to 2 from 4 would place the number `150` (original position 4) just before `90` (originally position 2) and increases the positions of both `90` and `120` by one, resulting in `[30, 60, 150, 90, 120, 180]`.

(a) (4.0 points)

Implement `promote`, which takes a list of numbers `s` and two non-negative integers `i` and `j`. It returns a new list promoting to `i` from `j` in `s`. Do not mutate `s`.

Hint: For any list `s`, `s[len(s):]` evaluates to `[]`.

```
def promote(s, i, j):
    """Return a list in which s[j] is at index i without mutating s.

    >>> promote([3, 6, 9, 12, 15, 18], 2, 4)
    [3, 6, 15, 9, 12, 18]
    >>> promote([3, 6, 9, 12, 15, 18], 0, 4)
    [15, 3, 6, 9, 12, 18]
    """
    assert i >= 0 and i < j and j < len(s)
    return s[:i] + _______ + _______ + s[_______:]
                    (a)       (b)          (c)
```

i. (2.0 pt) Fill in blank (a). **Select all correct answers.**

☐ `s[j]`

■ `[s[j]]`

☐ `list(s[j])`

☐ `s[j:j]`

■ `s[j:j+1]`

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

● `s[i:j]`

○ `s[i+1:j]`

○ `s[i:j+1]`

○ `s[i+1:j+1]`

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

○ `i`

○ `i+1`

○ `j`

● `j+1`

(b) (4.0 points)

Implement `promotions`, which takes two lists of numbers `s` and `t` that have the same elements, possibly in different orders. It returns the minimum number of promotions that must be applied to `s` so that it has the same order as `t`. Assume `promote` is implemented correctly.

```
def promotions(s, t):
    """Return the minimum times promote must be called to start from s and return t.

    >>> promotions([2, 4, 6, 8, 10, 12],
    ...            [2, 6, 8, 4, 12, 10])  # promote (1, 2) then (2, 3) then (4, 5)
    3
    >>> promotions([6, 1, 6, 1, 6, 1],
    ...            [1, 1, 6, 6, 1, 6])    # promote (0, 1) then (1, 5)
    2
    >>> promotions([1, 2, 3], [1, 2, 3])        # no promotions needed
    0
    >>> promotions([1, 2, 1, 2], [2, 1, 2, 1])  # promote (0, 3)
    1
    """
    assert sorted(s) == sorted(t)  # Check that s & t have the same elements (disregarding order)
    if len(s) == 0:
        return 0
    elif _______:
           (d)
        return promotions(s[1:], t[1:])
    else:
        return 1 + min([promotions(_______, t[1:]) for j in range(1, len(s)) if _______])
                                      (e)                                         (f)
```

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

- ○ `s == t`
- ○ `s != t`
- ● `s[0] == t[0]`
- ○ `s[0] != t[0]`
- ○ `sorted(s) == t`
- ○ `s == sorted(t)`

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

`promote(s, 0, j)[1:]` **OR** `s[:j] + s[j+1:]`

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

- ○ `s[0] == t[j]`
- ○ `s[1] == t[j]`
- ● `s[j] == t[0]`
- ○ `s[j] == t[1]`

(c) (8.0 points)

Implement `promote_link`, which takes a `Link` instance `s` (a non-empty linked list) and two non-negative integers `i` and `j` with `i<j` and `j` less than the length of `s`. It mutates `s` by promoting to `i` from `j` and then returns `s`.

The `Link` class appears on Page 2 (left side) of the Midterm 2 Study Guide.

```
def promote_link(s, i, j):
    """Mutate linked list s so that the item at index j is at index i.

    >>> a = Link(3, Link(6, Link(9, Link(12, Link(15, Link(18))))))
    >>> print(promote_link(a, 2, 4))
    <3 6 15 9 12 18>
    >>> print(promote_link(a, 0, 4))
    <12 3 6 15 9 18>
    >>> promote_link(a, 1, 3) is a
    True
    """
    assert i >= 0 and i < j
    if i > 0:
        _______
          (g)
    else:

        insert, tail = s.first, s.rest

        while j > 0:

            _______  # Hint: use multiple assignment: ___ , ___ = ___ , ___
              (h)

            tail, j = tail.rest, j-1

        _______ = insert
          (i)

    return _______
              (j)
```

i. (2.0 pt) Fill in blank (g).

- ○ `promote_link(s.rest, i-1, j)`
- ○ `return promote_link(s.rest, i-1, j)`
- ○ `promote_link(s.rest, i, j-1)`
- ○ `return promote_link(s.rest, i, j-1)`
- ● `promote_link(s.rest, i-1, j-1)`
- ○ `return promote_link(s.rest, i-1, j-1)`

ii. (2.0 pt) Fill in blank (h). **Hint**: Use multiple assignment: `___ , ___ = ___ , ___`

```
insert, tail.first = tail.first, insert
```

iii. (1.0 pt) Fill in blank (i).

● `s.first`

○ `s.rest.first`

○ `tail.first`

○ `tail.rest.first`

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

● `s`

○ `tail`

○ `Link(insert, tail.rest)`

○ `Link(s.first, tail.rest)`

v. (2.0 pt) What is displayed by the call to `print` in this code?

```
odds = Link(3, Link(5, Link(7, Link(9))))
for i in range(3):
    promote_link(odds, 0, 3)

print(odds)
```

○ `<3 5 7 9>`

○ `<3 9 7 5>`

● `<5 7 9 3>`

○ `<5 9 7 3>`

○ `<9 7 5 3>`

○ `<9 3 5 7>`

6. (12.0 points) Fresh Produce

(a) (5.0 points)

Implement `products`, which takes a `Tree` instance `t` with positive integer labels and a positive integer `n`. It returns `True` if every path from the root of `t` to a leaf has labels that equal `n` when multiplied together.

The `Tree` class appears on Page 2 (left side) of the Midterm 2 Study Guide.

```
def products(t, n):
    """Return whether the product of labels along every root-to-leaf path is n.

    >>> products(Tree(1, [Tree(2, [Tree(3)]), Tree(6)]), 6)
    True
    >>> products(Tree(1, [Tree(2, [Tree(3)]), Tree(6)]), 12)
    False
    >>> products(Tree(1, [Tree(2, [Tree(3)]), Tree(5)]), 6)
    False
    >>> products(Tree(1, [Tree(5, [Tree(2)]), Tree(12)]), 12)
    False
    """
    assert type(n) == int
    if t.is_leaf():
        return _______
                 (a)
    if _______:
        (b)

        return False

    return _______
             (c)
```

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

- ○ `True`
- ○ `False`
- ● `n == t.label`
- ○ `n % t.label == 0`
- ○ `n % t.label > 0`

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

- ○ `n != t.label`
- ○ `n < t.label`
- ○ `n > t.label`
- ● `n % t.label > 0`

iii. (3.0 pt) Fill in blank (c). **Important**: The second argument to a call to `products` must be an integer (`int`).

```
all([products(b, n // t.label) for b in t.branches])
```

(b) (7.0 points)

Definition. An *increasing sequence* is a sequence of integers in which each element after the first is larger than the previous element.

Implement `produce`, which takes a positive integer `n`. It returns a `Tree` of positive integers in which:

- The product of the labels along every root-to-leaf path is `n`,
- Every increasing sequence of integers starting with 1 that has product `n` is a root-to-leaf path, and
- Every sequence of siblings (nodes with a common parent) is an increasing sequence.

```
def produce(n):
    """Return the largest tree in which the labels for every root-to-label path
    are increasing and have product n. Put all siblings in increasing order.

    >>> produce(12)
    Tree(1, [Tree(2, [Tree(6)]), Tree(3, [Tree(4)]), Tree(12)])
    >>> print(produce(24))  # Paths are 1-2-3-4, 1-2-12, 1-3-8, 1-4-6, and 1-24
    1
      2
        3
          4
        12
      3
        8
      4
        6
      24
    """
    def grow(t, x):

        for k in range(_______, x + 1):
                        (d)

            if _______ % k == 0:
                 (e)

                branch = _______
                          (f)

                if _______:
                     (g)

                    t.branches.append(branch)
        return t

    return grow(Tree(1), n)
```

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

- ○ `0`
- ○ `1`
- ○ `x`
- ○ `t.label`
- ● `t.label + 1`

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

- ● `x`
- ○ `n`
- ○ `t.label`
- ○ `(t.label // x)`
- ○ `(t.label // k)`

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

- ○ `Tree(k)`
- ○ `Tree(x)`
- ○ `grow(Tree(k), x)`
- ○ `grow(Tree(x), k)`
- ● `grow(Tree(k), x // k)`
- ○ `grow(Tree(x), x // k)`

iv. (3.0 pt) Fill in blank (g).

```
not branch.is_leaf() or x == k
```

7. (14.0 points) A Pair of Schemes

(a) (4.0 points)

Implement the Scheme procedure `all-pairs`, which takes a procedure `f` and a list `s`. It returns `#t` if `(f x y)` is `#t` for every pair of adjacent elements `x` and `y` in `s`. Assume `f` always returns either `#t` or `#f`.

```
(define (inc x y) (= (+ x 1) y))  ; Whether x+1 equals y

;;; Return #t if (f x y) is #t for every pair of adjacent values (x, y) in list s.
;;;
;;; scm> (all-pairs inc '(3 4 5 6 7 8))
;;; #t
;;; scm> (all-pairs inc '(3 4 5 8 7 8))
;;; #f
;;; scm> (all-pairs inc '(3))
;;; #t
(define (all-pairs f s)
  (or (null? s) (null? (cdr s))
      (and _______ (all-pairs f _______ ))))
              (a)                 (b)
```

i. (3.0 pt) Fill in blank (a). **Select all correct answers.**

- [ ] `(car s) (cdr s)`
- [ ] `(car s) (car (cdr s))`
- [ ] `f (car s) (cdr s)`
- [ ] `f (car s) (car (cdr s))`
- [ ] `f((car s) (cdr s))`
- [ ] `f((car s) (car (cdr s)))`
- [ ] `(f (car s) (cdr s))`
- [x] `(f (car s) (car (cdr s)))`
- [ ] `(apply f (car s) (cdr s))`
- [ ] `(apply f (car s) (car (cdr s)))`

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

- ○ `s`
- ● `(cdr s)`
- ○ `(cdr (cdr s))`
- ○ `(cons (car s) (cdr s))`
- ○ `(cons (car s) (cdr (cdr s)))`

(b) (6.0 points)

Implement the Scheme procedure `show-pairs`, which takes a list `s` and returns a list of every pair of adjacent elements in `s`. A pair is a two-element list.

```
;;; Return a list of every pair of adjacent elements in list s.
;;;
;;; scm> (show-pairs '(3 5 7 9 11 13))
;;; ((3 5) (5 7) (7 9) (9 11) (11 13))
(define (show-pairs s)
  (if (or (null? s) (null? (cdr s))) nil
    ( _______ _______ (show-pairs _______ ))))
        (a)     (b)                 (c)
```

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

- ○ `car`
- ○ `cdr`
- ● `cons`
- ○ `list`
- ○ `append`

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

```
(list (car s) (car (cdr s)))
```

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

- ○ `s`
- ● `(cdr s)`
- ○ `(cdr (cdr s))`
- ○ `(cons (car s) (cdr s))`
- ○ `(cons (car s) (cdr (cdr s)))`

iv. (1.0 pt) What order of growth describes the time it takes to execute `(unpair s)` in terms of the length of the input list `s`, assuming that `car`, `cdr`, `cons`, and `null?` are all constant-time operations?

```
(define (unpair s) (cond ((null? s) nil)
                         ((null? (cdr s)) (car s))
                         (else (cons (car (car s)) (unpair (cdr s))))))
```

- ○ constant
- ○ logarithmic
- ● linear
- ○ quadratic
- ○ exponential

(c) (4.0 points)

Implement the Scheme procedure `all-pairs-exp`, which takes a procedure name `proc-name` (a symbol) and a list `s`. It returns an `and` expression that calls the procedure named by `proc-name` on every adjacent pair of elements in `s`. Assume `show-pairs` is implemented correctly.

```
;;; Return an and expression that calls the procedure called proc-name on
;;; every adjacent pair of elements in s.
;;;
;;; scm> (all-pairs-exp 'inc '(3 4 5 6 7 8))
;;; (and (inc 3 4) (inc 4 5) (inc 5 6) (inc 6 7) (inc 7 8))
;;; scm> (eval (all-pairs-exp 'inc '(3 4 5 6 7 8 )))
;;; #t
(define (all-pairs-exp proc-name s)
  ( _______ (map _______ (show-pairs s))))
     (a)           (b)
```

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

- ○ `and`
- ○ `'and`
- ○ `cons and`
- ● `cons 'and`

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

`(lambda (pair) (cons proc-name pair))` **or** `(lambda (pair) (list proc-name (car pair) (car (cdr pair))))`

8. (5.0 points) Arcane Jobs

The Council of Piltover wants a list of all the people living in regions it governs that have unique jobs. Create a table with columns labeled `name` and `job` that contains one row for each person living in a region governed by the `"council"` who is the only person with their job among everybody living in regions ruled by the council.

The `who` table has one row per person and columns for their `name` (`string`; each row has a unique value), the `region` (`string`) they live in, and their `job` (`string`). The `gov` table has one row per region and columns for its name called `place` (`string`; each row has a unique value) and the group that rules over it called `ruler` (`string`).

Here's an example of the contents of the `who` and `gov` tables and the expected result for a query based on these.

who:

name	region	job
caitlyn	piltover	enforcer
vi	zaun	enforcer
jayce	piltover	scientist
victor	piltover	scientist
ekko	zaun	firelight
heimerdinger	piltover	scientist
jinx	zaun	rebel
mel	noxus	councilor
ambessa	noxus	warlord
vander	zaun	barkeep
silco	zaun	chembaron

gov:

place	ruler
noxus	triumvirate
zaun	council
piltover	council
bilgewater	gangs

expected result:

name	job
vander	barkeep
silco	chembaron
ekko	firelight
jinx	rebel

```
SELECT name, job FROM _______ WHERE _______ AND ruler = "council" _______ ;
                        (a)             (b)                              (c)
```

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

- ○ `who`
- ○ `who AS a, who AS b`
- ● `who, gov`
- ○ `who AS a, who AS b, gov`

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

- ○ `COUNT(*) = 1`
- ○ `place = "council"`
- ○ `a.job = b.job`
- ○ `a.job != b.job`
- ● `region = place`
- ○ `a.region = place`
- ○ `a.region = b.region AND a.region = place`
- ○ `a.job = b.job AND a.region = place`

(c) **(2.0 pt)** Fill in blank (c). You may write `AND` to continue the `WHERE` clause (but you don't have to). You may also include other clauses such as `GROUP BY`, `ORDER BY`, `HAVING`, and `LIMIT` (but you don't have to).

```
GROUP BY job HAVING COUNT(*) = 1
```

9. (0.0 points) Ape Pull Us

These two A+ questions are not worth any points. They can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first!

(a) Implement `all-next`, a macro that takes an expression `x-expr` and a list of numbers. It returns `#t` if every element in `s` besides the first is equal to the value of `x-expr` when `x` is bound to the previous element in `s`. It returns `#f` otherwise. Assume `x-expr` does not contain the symbol `y`.

You may call `all-pairs` from Q6. **Important:** The template has a quasiquote before the blank.

```
;;; Return whether every value in s that follows another value x is equal
;;; to the x-expr evaluated when x is the previous value.
;;;
;;; scm> (all-next (+ x 2) '(3 5 7 9 11 13))
;;; #t
;;; scm> (all-next (* x 2) '(2 4 8 16 32 64))
;;; #t
;;; scm> (all-next (+ x 2) '(3 5 7 8 11 13))
;;; #f
;;; scm> (all-next (+ x 4) '(3 5 7 9 11 13))
;;; #f
(define-macro (all-next x-expr s) ` _______ )
```

```
(all-pairs (lambda (x y) (= ,x-expr y)) ,s)
```

(b) Implement `make_tens` by filling in the blank in `repromote`. The `make_tens` function takes a list of numbers `s`. It returns a list `t` of the same elements reordered so that `tens(t)` returns `True`. Return an order that requires the fewest promotions to reorder `s` into `t`. You may use `tens` from Q2 and `promote` from Q4.

```
def make_tens(s):
    """Return a list t for which tens(t) is true and promotions(s, t) is as small as possible.
    If there is no reordering t of s for which tens(t) is true, return None.
    >>> make_tens([4, 2, 2, 2, 4, 6, 1, 3, 3, 2, 5]) # promote (6, 7), (7, 9), and (8, 10)
    [4, 2, 2, 2, 4, 6, 3, 2, 5, 1, 3]
    >>> make_tens([4, 2, 2, 2, 4, 5, 1, 4, 3, 4]) # promote (2, 4), (4, 7), (5, 9), and (8, 9)
    [4, 2, 4, 2, 4, 4, 2, 5, 3, 1]
    """
    for t in promute(s):
        if tens(t):
            return t
def promute(s):
    for k in range(len(s)):
        yield from repromote(s, k)
def repromote(s, k):
    if k == 0:
        yield s
    elif len(s) > 1:
        for j in range(len(s)):
            for rest in _______:
                yield [s[j]] + rest
```

```
repromote(s[:j] + s[j+1:], k - min(1, j))
```

No more questions.