CS 61A
Summer 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. **(8.0 points)** **Significant Factors**

   (a) **(3.0 points)**

   Implement `significant`, which takes positive integers `n` and `k`. It returns the `k` *most significant* digits of `n` as an integer. These are the first `k` digits of `n`, starting from the left. If `n` has fewer than `k` digits, it returns `n`. You may not use `round`, `int`, `str`, or any functions from the `math` module.

   You may use `pow`, which raises its first argument to the power of its second: `pow(9, 2)` is 81 and `pow(9, 0.5)` is 3.0.

   ```python
   def significant(n, k):
       """Return the K most significant digits of N.

       # Case 1
       >>> significant(12345, 3)
       123

       # Case 2
       >>> significant(12345, 7)
       12345
       """

       if _____:
   #                   (a)

           return n

       return significant(_____, _____)
   #                          (b)              (c)
   ```

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

   > **n < pow(10, k)**

   ii. **(1.0 pt)** Which of these could fill in blank (b)?
   - ○ k - 1
   - ○ k / 10
   - ○ k // 10
   - ○ k % 10
   - ○ n - 1
   - ○ n / 10
   - ● n // 10
   - ○ n % 10

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

   > **k**

(b) **(5.0 points)**

Implement `factorize`, which takes two integers `n` and `k`, both larger than 1. It returns the number of ways that `n` can be expressed as a product of non-decreasing integers greater than or equal to `k`.

```
def factorize(n, k=2):
    """Return the number of ways to factorize positive integer n.

    # Case 1
    >>> factorize(7) # 7
    1

    # Case 2
    >>> factorize(12) # 2*2*3, 2*6, 3*4, 12
    4

    # Case 3
    >>> factorize(36) # 2*2*3*3, 2*2*9, 2*3*6, 2*18, 3*3*4, 3*12, 4*9, 6*6, 36
    9
    """

    if n == k:

        return 1

    elif _____:
#               (a)

        return 0

    elif _____:
#               (b)

        return factorize(n, k + 1)

    return _____
#               (c)
```

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

> $k > n$

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

- ◯ `n == k`
- ◯ `n > k`
- ◯ `n < k`
- ◯ `n % k == 0`
- ⬤ `n % k > 0`
- ◯ `n % k < 0`

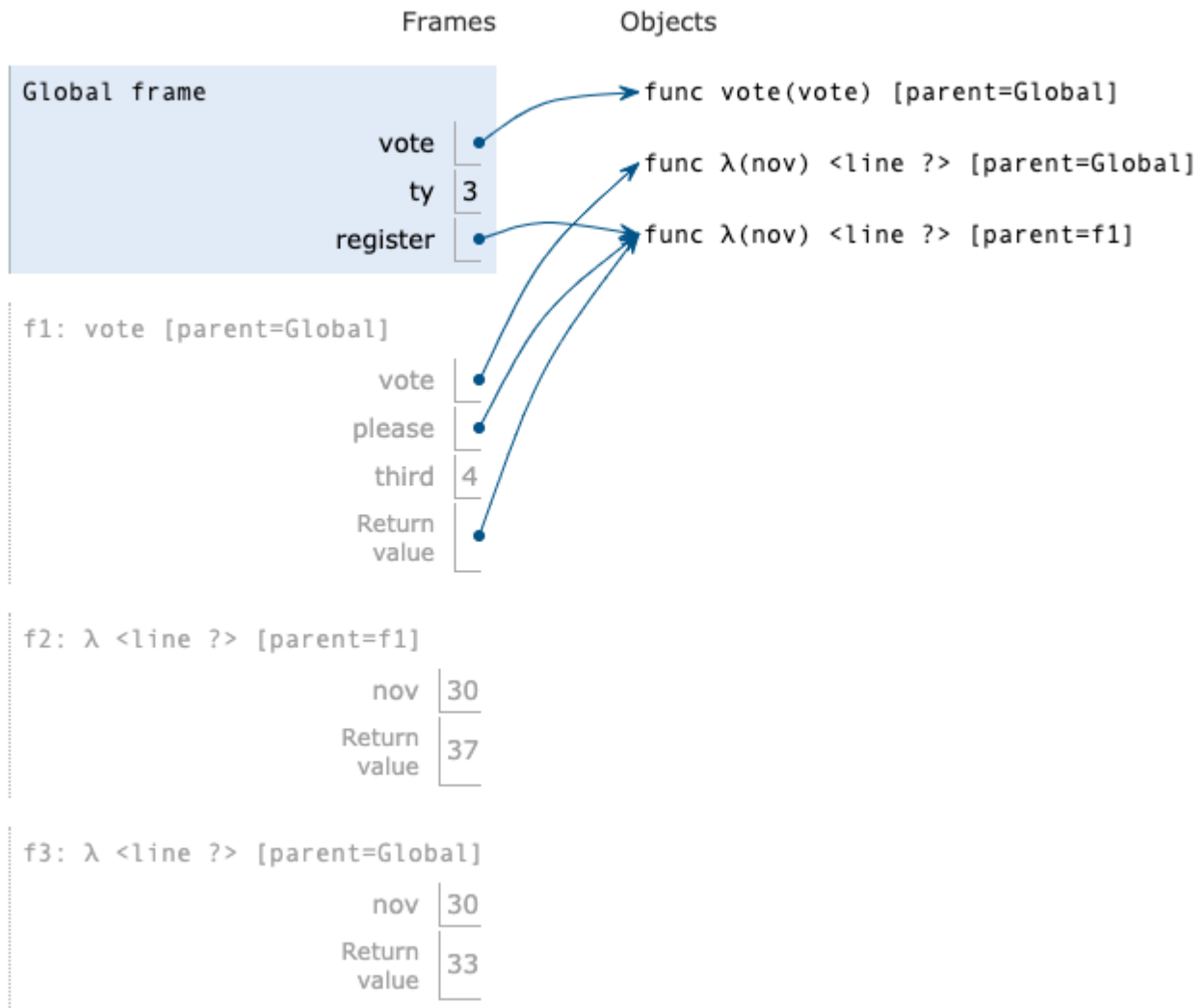**iii. (2.0 pt)** Fill in blank (c).

factorize(n//k, k) + factorize(n, k + 1)

2. (8.0 points)   Please Register to Vote

Fill in each blank in the code example below so that its environment diagram is the following.

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

Frames   Objects

Global frame   → func vote(vote) [parent=Global]

                    vote •   func λ(nov) <line ?> [parent=Global]

                     ty 3

                register •   func λ(nov) <line ?> [parent=f1]

f1: vote [parent=Global]

                    vote •

                  please •

                   third 4

                  Return •
                   value

f2: λ <line ?> [parent=f1]

                     nov 30

                  Return 37
                   value

f3: λ <line ?> [parent=Global]

                     nov 30

                  Return 33
                   value

```
def vote(vote):

    please = _____
#                 (a)

    _____ = ty + 3
#       (b)

    return please

ty = 1

register = _____(lambda nov: nov + ty)
#                (c)
```

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

          register(_____)
          #              (e)
```

(a) **(2.0 pt)** Which of these could fill in blank (a)?

  ○ `vote(ty)`

  ○ `vote(30)`

  ○ `vote`

  ● `lambda nov: vote(nov) + third`

  ○ `lambda nov: vote(nov + third)`

  ○ `lambda nov: vote(nov) + ty`

  ○ `lambda nov: vote(nov + ty)`

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

  ● `third`

  ○ `ty`

  ○ `please`

  ○ `vote`

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

  ○ `third`

  ○ `ty`

  ○ `please`

  ● `vote`

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

> `ty = 3`

(e) **(2.0 pt)** Which of these could fill blank (e)? **Check all that apply.**

  ■ `ty * 10`

  □ `ty - 1 + 30`

  ■ `30`

  □ `third + 26`

  □ `(lambda x: x + x)(15)`

3. (10.0 points)   Amazing Job Growth

Definition. A *repeatable* function is a function that returns a repeatable function.

Reminder. You may call built-in functions that do not require import, such as `min`, `max`, `abs`, and `pow`.

(a) **(4.0 points)**

Implement `growth`, which takes a number `baseline` and returns a repeatable function `increase`. When `increase` is called on a number `observed`, it prints the difference between `observed` and the smallest argument passed to `growth` or `increase` so far among the repeated calls.

```python
def growth(baseline):
    """Return a function that can be called repeatedly on numbers and prints
    the difference between its argument and the smallest argument used so far
    (including baseline).


    # Case 1
    >>> job = growth(148)(149)(150)(130)(133)(139)(137)
    1
    2
    0
    3
    9
    7
    """
    def increase(observed):

        under = _____
#                   (a)

        print(observed - under)

        return _____
#                   (b)

    return increase
```

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

> **min(observed, baseline)**

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

○ `increase`

○ `increase(under)`

○ `increase(observed)`

○ `increase(baseline)`

○ `growth`

● `growth(under)`

○ `growth(observed)`

○ `growth(baseline)`

(b) (6.0 points)

Implement `maxer`, a higher-order function that takes a function `smoke`, which takes a number and returns a number. The `maxer` function returns a repeatable function `fire` that takes a number y and prints the largest result of calling `smoke` on any value of y passed to `fire` so far among the repeated calls.

Assume that `smoke` is a deterministic pure function.

```python
def square(x):
    return x * x

def maxer(smoke):
    """Return a repeatable function fire(y) that prints the largest smoke(y) so far.

    # Setup
    >>> g = maxer(square)

    # Case 1
    >>> h = g(2)(1)(3)(2)(-4)  # print the largest square(y) so far
    4
    4
    9
    9
    16

    # Case 2
    >>> h = maxer(abs)(2)(1)(3)(2)(-4)  # print the largest abs(y) so far
    2
    2
    3
    3
    4
    """

    def fire(y):


        _____
#           (a)

        def haze(z):

            if _____:
#                 (b)

                z = y

            return _____
#                      (c)

        return haze

    return fire
```

i. (2.0 pt) Fill in blank (a). You may not write a `return` statement for this blank.

> **print(smoke(y))**

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

> **smoke(y) > smoke(z)**

iii. (2.0 pt) Which of these could fill in blank (c)?

- ○ `y`
- ○ `smoke(y)`
- ○ `fire(y)`
- ○ `fire(smoke(y))`
- ○ `haze`
- ○ `haze(y)`
- ○ `haze(smoke(y))`
- ○ `z`
- ○ `smoke(z)`
- ● `fire(z)`
- ○ `fire(smoke(z))`
- ○ `haze(z)`
- ○ `haze(smoke(z))`

No more questions.