CS 61A Fall 2020

Structure and Interpretation of Computer Programs

Midterm 1 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?

1. (14.0 points) Down for the Count

Definition. A *digit* is a non-negative integer less than 10. Integers contain digits.

Examples.

- The integer 21 contains the digits 1 and 2.
- The integer 474 contains the digit 4 twice and the digit 7 once.
- The integer 400 contains the digit 4 once and the digit 0 twice.
- The integer -77 contains the digit 7 twice.
- The integer 0 is a 0-digit number that contains no digits.

Reminders.

- You may call built-in functions that do not require import, such as `min`, `max`, `abs`, and `pow`.
- You may call functions defined in earlier parts of the question in your implementation for later parts, and you may assume that the functions you call are implemented correctly.

RESTRICTION. You may not call `str` or `repr` or use `[` or `]` in any part of this question.

(a) (4.0 points)

Implement `count`, which takes a digit `element` and an integer `box`. It returns the number of times that `element` appears in `box`.

Warning: `n % d` and `n // d` may not behave as you expect for negative `n`. For example, `-123 % 10` evaluates to 7. `-1 // 10` evaluates to -1. You do not need to know how these operators apply to negative `n` in order to solve this problem.

```
def count(element, box):
    """Count how many times digit element appears in integer box.

    >>> count(2, 222122)
    5
    >>> count(0, -2020)
    2
    >>> count(0, 0)  # 0 has no digits
    0
    """
    assert element >= 0 and element < 10


    _________
       (a)

    total = 0

    while box > 0:

        if _________:
              (b)

            total = _________
                       (c)

        box = box // 10

    return total
```

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

```
box = abs(box)
```

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

- ○ `box == element`
- ● `box % 10 == element`
- ○ `box % element == 0`
- ○ `box % element > 0`

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

- ● `total + 1`
- ○ `element`
- ○ `total + element`
- ○ `box % 10`
- ○ `total + box % 10`

(b) (5.0 points)

Implement `count_nine`, which takes a digit `element` and a non-negative integer `box`. It returns the number of times that `element` appears in `box` and is not adjacent to a 9.

```
def count_nine(element, box):
    """Count how many times digit element appears in the non-negative integer
    box in a place that is not next to a 9.

    >>> count_nine(2, 222122)
    5
    >>> count_nine(1, 1911191) # Only the middle 1 is not next to a 9
    1
    >>> count_nine(9, 9)
    1
    >>> count_nine(9, 99)
    0
    >>> count_nine(3, 314159265359)
    2
    >>> count_nine(5, 314159265359)
    1
    >>> count_nine(9, 314159265359)
    2
    >>> count_nine(0, 0) # No digits are in 0
    0
    """
    assert element >= 0 and element < 10
    assert box >= 0

    nine, total = False, 0

    while box > 0:

        if _________ and not (nine or _________):
              (a)                       (b)

            total = _________
                      (c)

        nine = _________ == 9
                  (d)

        box = box // 10

    return total
```

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

- ○ `box == element`
- ● `box % 10 == element`
- ○ `box % element == 0`
- ○ `box % element > 0`

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

(box // 10) % 10 == 9

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

- (●) `total + 1`
- ( ) `element`
- ( ) `total + element`
- ( ) `box % 10`
- ( ) `total + box % 10`

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

box % 10

(c) (5.0 points)

Implement `fit`, which takes two non-negative integers `pegs` and `holes`. It returns whether every digit in `pegs` appears at least as many times in `holes` as it does in `pegs`.

```
def fit(pegs, holes):
    """Return whether every digit in pegs appears at least as many times in
    holes as it does in pegs.

    >>> fit(123, 321)    # Each digit appears once in pegs and in holes.
    True
    >>> fit(1213, 33221) # 1 appears twice in pegs, but only once in holes.
    False
    >>> fit(12, 22)      # 1 appears once in pegs, but not at all in holes.
    False
    >>> fit(314159, 112233456789)
    True
    """
    i = 0

    while i <= _________:
                  (a)

        if _________:
              (b)

            _________
               (c)

        i = i + 1

    return _________
              (d)
```

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

9

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

count(i, pegs) > count(i, holes)

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

return False

iv. (1.0 pt) Which of these could fill in blank (d)?

- (●) `True`
- ( ) `False`
- ( ) `holes > pegs`
- ( ) `pegs > holes`
- ( ) `holes >= pegs`
- ( ) `pegs >= holes`

2. (8.0 points) Mystery Function

Assume `mystery` is a deterministic pure function that takes one integer argument, returns an integer, and never errors.

```
def mystery(n):
    ...
```

Assume the following functions are also defined:

```
def add_two(y):
    return y + 2

def two(y):
    return 2

def constant(k):
    def ignore(x):
        return k
    return ignore

def diff(f, g):
    return lambda z: abs(f(z) - g(z))
```

Definition. Two functions `f` and `g` have identical behavior if `f(x)` and `g(x)` return equal values or return functions with identical behavior, for every `x` that does not cause an error.

Complete each statement below so that it is true for all possible deterministic pure `mystery` functions.

(a) (2.0 pt) The result of evaluating `constant(2)` has identical behavior to the result of evaluating the expression...

- ◯ ... `add_two`
- ◯ ... `add_two(0)`
- ◯ ... `add_two(2)`
- ● ... `two`
- ◯ ... `two(0)`
- ◯ ... `two(2)`
- ◯ None of these

(b) (2.0 pt) The result of evaluating `diff(constant(1), constant(-1))` has identical behavior to the result of evaluating the expression...

- ◯ ... `constant`
- ◯ ... `constant(0)`
- ● ... `constant(2)`
- ◯ ... `diff(constant, constant)`
- ◯ None of these

(c) (2.0 pt) The result of evaluating `diff(mystery, mystery)` has identical behavior to the result of evaluating the expression...

- ◯ ... `constant`
- ● ... `constant(0)`
- ◯ ... `constant(2)`
- ◯ ... `diff(constant, constant)`
- ◯ ... `constant(mystery)`
- ◯ ... `mystery`
- ◯ None of these

(d) (2.0 pt) The result of evaluating `diff(mystery, diff(mystery, mystery))` has identical behavior to the result of evaluating the expression...

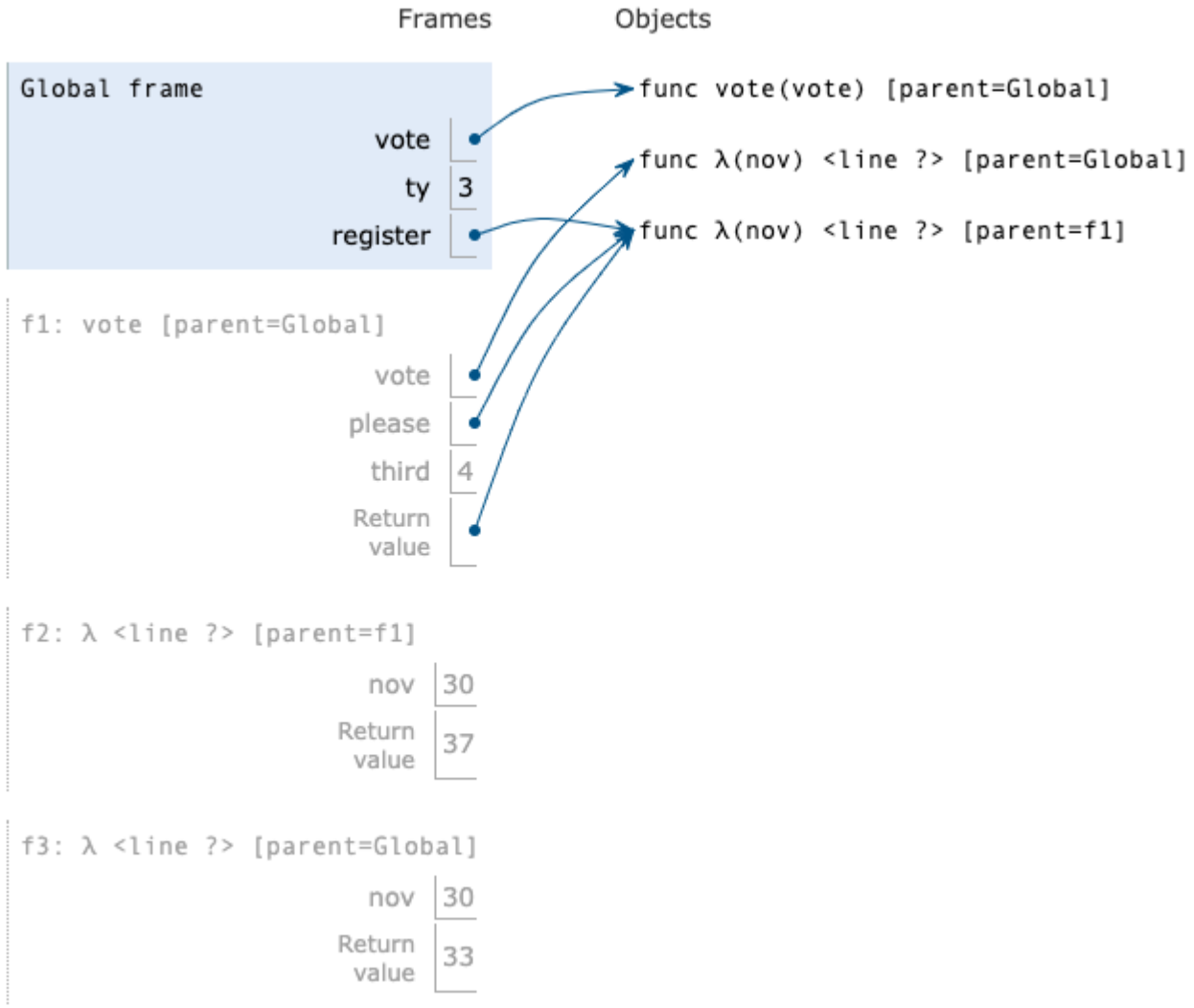
- ◯ ... `mystery`
- ◯ ... `abs(mystery)`
- ● ... `lambda y: abs(mystery(y))`
- ◯ ... `lambda y: mystery(abs(y))`
- ◯ ... `lambda y: lambda z: mystery(abs(y)) - mystery(abs(z))`
- ◯ ... `lambda y: lambda z: abs(mystery(y)) - abs(mystery(z))`
- ◯ ... `lambda y: lambda z: abs(mystery(y) - mystery(z))`
- ◯ None of these

3. (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




```
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)`

4. (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.

```
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).

    >>> 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.

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

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

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