CS 61A Structure and Interpretation of Computer Programs

Fall 2022 MIDTERM 1

INSTRUCTIONS

You have 1 hour and 50 minutes to complete the exam.

- The exam is closed book, closed notes, closed computer, closed calculator, except one 8.5" × 11" page of your own creation and the provided midterm 1 study guide.
- Mark your answers on the exam itself in the spaces provided. We will not grade answers written on scratch paper or outside the designated answer spaces.
- If you need to use the restroom, bring your phone and exam to the front of the room.
- You may use built-in Python functions that do not require import, such as `pow`, `len`, `abs`, `bool`, `int`, `float`, `str`, `max`, and `min`.
- You **may not** use example functions defined on your study guide unless a problem clearly states you can.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.
- You may not use ; to place two statements on the same line.

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!

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?

1. (4.0 points) What Would Python Display?

Assume the following code has been executed.

```
bear = -1
oski = lambda print: print(bear)
bear = -2

s = "Knock"
```

For each expression below, write the output **displayed by the interactive Python interpreter** when the expression is evaluated. The output may have multiple lines. If an error occurs, write "Error", but include all output displayed before the error. If evaluation would run forever, write "Forever". To display a function value, write "Function". The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

(a) (1.0 pt) `(3 and 4) - 5`

- ○ -2
- ● -1
- ○ 1
- ○ 2
- ○ Error

The `and` operator returns the first falsey value (a false value, but not necessarily `False`), or the last value if none are falsey. Since 3 is truthy (a true value, even though it's not `True`), `3 and 4` evaluates to 4. `4 - 5` evaluates to -1.

(b) (1.0 pt) `oski(abs)`

- ○ -2
- ○ -1
- ○ 1
- ● 2
- ○ Function
- ○ Error

The lambda function bound to `oski` takes a function and calls it on `bear`. Thus, `oski(abs)` evaluates `abs(bear)`. Since `bear` isn't defined in the lambda function, we take the global value of `bear`, which is -2. `abs(-2)` is 2.

(c) (2.0 pt) `print(print(print(s, s) or print("Who's There?")), "Who?")`

Knock Knock
Who's There?
None
None Who?

Python evaluation order evaluates the operator expression (to a function), then the operands from left to right (to the arguments), then calls the function on its arguments.

Below, we substitue operands with argument values and describe the side effects of doing so.

`print(print(print(s, s) or print("Who's There?")), "Who?")`

`print(print(None or print("Who's There?")), "Who?")` ; Calls print on "Knock Knock"

`print(print(print("Who's There?")), "Who?")` ; Since `None` is a false value, `None or print(...)` evaluates the `print` call.

`print(print(None), "Who?")` ; Calls print on "Who's There?"

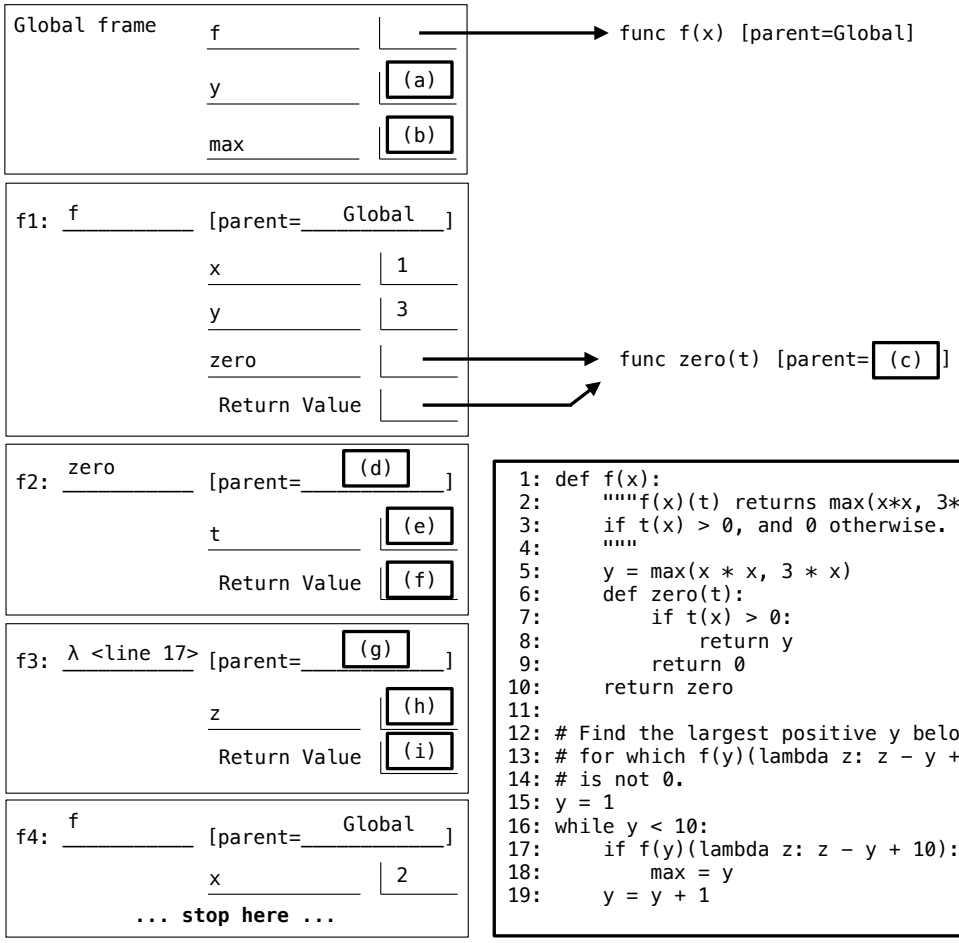
`print(None, "Who?")` ; Calls print on None

`None` ; Calls print on two arguments and displays "None Who?"

Since the value of the whole expression is None, that final None is not displayed.

2. (12.0 points) Don't Call Me, I'll Call You

When working on a large project, you get the error `'int' object is not callable` in the code below during the second call to `f`. Complete the environment diagram until the error occurs to identify the problem with the code. **If a blank contains an arrow to a function, write the function as it would appear in the diagram.**




```
 1: def f(x):
 2:     """f(x)(t) returns max(x*x, 3*x)
 3:     if t(x) > 0, and 0 otherwise.
 4:     """
 5:     y = max(x * x, 3 * x)
 6:     def zero(t):
 7:         if t(x) > 0:
 8:             return y
 9:         return 0
10:     return zero
11:
12: # Find the largest positive y below 10
13: # for which f(y)(lambda z: z - y + 10)
14: # is not 0.
15: y = 1
16: while y < 10:
17:     if f(y)(lambda z: z - y + 10):
18:         max = y
19:     y = y + 1
```

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

2

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

1

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

- ○ `Global`
- ○ `f`
- ● `f1`

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

- ○ `Global`
- ○ `f`
- ● `f1`

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

func λ(z) <line 11> [parent=Global]

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

3

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

- ● `Global`
- ○ `f`
- ○ `zero`
- ○ `f1`
- ○ `f2`

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

1

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

10

(j) (2.0 pt) Explain in 10 words or less why the `'int' object is not callable` error occurred.

Example answer: "Global `max` was reassigned to an integer"

The first iteration of the while loop works as intended; `f(y)` returns the function `zero` with `x` set to `1` and `y` set to 3. We then call `zero` on `lambda z: z - y + 10`. This evaluates `(lambda z: z - y + 10)(1) == 1 - 1 + 10 == 10`. Since `10 > 0`, we return `y`, which is equal to `3`. This is truthy, so we set `max` to `1`.

On the second iteration of the loop, we run until line 5. At this point, we evaluate a call expression with `max` as the operator. However, `max` was set to 1 in the previous iteration, overwriting its original assignment to the built-in `max` function. Thus, Python tries to call `1` as a function; since `1` is not a function, Python errors, stating that `'int' object is not callable`.

3. (10.0 points) It's Perfect

Definitions. A *perfect* number is a positive integer `n` whose *proper factors* (the factors of `n` below `n`) sum to exactly `n`. A number `n` is *abundant* if the sum of `n`'s proper factors is greater than `n` and *deficient* if that sum is less than `n`.

(a) (4.0 points)

Implement `classify`, a function that takes an integer `n` **greater than** 1. It returns the string `'deficient'`, `'perfect'`, or `'abundant'` that correctly describes `n`.

```
def classify(n):
    """Return whether n > 1 is 'deficient', 'perfect', or 'abundant'.

    >>> classify(6)   # Proper factors 1, 2 and 3 sum to exactly 6.
    'perfect'
    >>> classify(24)  # Proper factors 1, 2, 3, 4, 6, 8, and 12 sum to 36.
    'abundant'
    >>> classify(23)  # Proper factor 1 sums to 1.
    'deficient'
    """
    total, k = 0, 1
    while __<BLANK 1>__:
        if __<BLANK 2>__:
            __<BLANK 3>__
        k = k + 1
    if total == n:
        return 'perfect'
    elif __<BLANK 4>__:
        return 'deficient'
    else:
        return 'abundant'
```

i. (1.0 pt) BLANK 1

- (●) `k < n`
- ( ) `k <= n`
- ( ) `k < total`
- ( ) `k <= total`

ii. (1.0 pt) BLANK 2

n % k == 0

iii. (1.0 pt) BLANK 3

total += k

iv. (1.0 pt) `BLANK 4`

total < n

In order to classify a number, we divide the code roughly into two steps:

- Compute the sum of all the proper factors of `n`
- Compare that sum to `n` to determine what classification to return

The first step is done with the `while` loop; we iterate over all values of `k` from 1 to `n` (not including `n`), and check if each one is a factor of `n`. This can be done by running `n % k`; if this is zero, then `k` divides `n` evenly, so `k` is a factor of `n`. If `k` is a factor of `n`, we add its value to the running total.

The second step is done by checking if `total` is either equal to, less than, or greater than `n`. This is done in the last `if/elif/else` clause; since we want to return `'deficient'` when `total < n`, we pick that to fill blank 4.

(b) (6.0 points)

Implement `nearest_perfect`, which takes an integer `n` above 5. It returns the nearest perfect number to `n`. If two perfect numbers are equally close to `n`, return the larger one. A number `a` is nearer to `n` than another number `b` if `abs(a - n) < abs(b - n)`. Assume `classify` is implemented correctly.

```
def nearest_perfect(n):
    """Return the nearest perfect number to n. In a tie, return the larger one.

    >>> nearest_perfect(8)   # 6 is perfect and 2 away from 8.
    6
    >>> nearest_perfect(20)  # 28 is perfect and 8 away from 20.
    28
    >>> nearest_perfect(17)  # Both 6 and 28 are 11 away from 17.
    28
    >>> nearest_perfect(6)   # 6 is perfect and 0 away from 6.
    6
    """
    k = 0
    while True:
        if __<BLANK 5>__:
            __<BLANK 6>__
        if __<BLANK 7>__:
            k = -k
        else:
            __<BLANK 8>__
```

i. (2.0 pt) `BLANK 5`

classify(n+k) == "perfect"

ii. (1.0 pt) `BLANK 6`

- ○ `n = n + k`
- ○ `n = n + 1`
- ○ `n = n - k`
- ○ `n = n - 1`
- ● `return n + k`
- ○ `return n`
- ○ `return k`

iii. (1.0 pt) `BLANK 7`

k > 0; Alternate solution: classify(n-k)=="perfect"

iv. (2.0 pt) `BLANK 8`

k = -k + 1; Alternate solution: k = k+1

The key observation here is that we can determine if a number `k` is perfect by checking if `classify(k)` returns `'perfect'`.

We can further note that according to the rules for returning, we should return the first perfect number that appears in the sequence `n, n+1, n-1, n+2, n-2, ...`

For example, if we try to get `nearest_perfect(17)`, we should check, in order, `17, 18, 16, 19, 15, 20, ..., 28, 6, ...`. Among these values, `28` is the first perfect number, so it is the nearest perfect number to 17. Note that we check `17 + 11 == 28` before we check `17 - 11 == 6`, so we return the larger perfect number if `n` is equally close to two perfect numbers.

Thus, we can fill blanks 5 and 6 so check if `n+k` is a perfect number, and return `n+k`, respectively. For blanks 7 and 8, we want to make it so that `k` goes `0, 1, -1, 2, -2, ...`. We note that after every positive number `k` in this sequence, the next number is `-k`, so we set blank 7 to check if `k > 0`. For all other numbers, we go from `0 -> 1`, `-1 -> 2`, `-2 -> 3`, and so on. This can be done by negating `k` and adding 1; that is, `1 == -0 + 1`, `2 == -(-1)+1`, etc. Thus we fill blank 8 with `k = -k + 1`.

An alternative solution is to check if `classify(n-k)=='perfect'` in line 7. If this is the case, then k gets set to -k, and we return the appropriate perfect number in the next iteration of the while loop. Using this, line 8 needs only to increment k, so we can do `k += 1` on that line.

4. (4.0 points) Super Powers

Definition. A *function power*, written $f^n(x)$, describes repeated application of a function f to x. $f^2(x) = f(f(x))$, $f^5(x) = f(f(f(f(x))))$, and so on.

(a) (2.0 pt) Choose **all** correct implementations of `funsquare`, a function that takes a one-argument function `f`. It returns a one-argument function `f2` such that `f2(x)` has the same behavior as `f(f(x))` for all `x`.

```
>>> triple = lambda x: 3 * x
>>> funsquare(triple)(5)  # Equivalent to triple(triple(5))
45
```

```
A:  def funsquare(f):
        return f(f)
```

```
B:  def funsquare(f):
        return lambda: f(f)
```

```
C:  def funsquare(f, x):
        def g(x):
            return f(f(x))
        return g
```

```
D:  def funsquare(f):
        return lambda x: f(f(x))
```

```
E:  def funsquare(f, x):
        return f(f(x))
```

```
F:  def funsquare(f):
        def g(x):
            return f(f(x))
        return g
```

☐ A

☐ B

☐ C

■ D

☐ E

■ F

C and E receive two inputs, while `funsquare` is defined to only receive one input; thus, these two are incorrect.

For A and B, we end up calling `f` with a function as an argument, even if `f` expects to receive an integer as an argument. Thus, these two cannot work.

D and F both work; we can verify this by trying to run these functions on the doctests.

(b) (1.0 pt) Fill in the blank to assign `quadruple` to a function that takes a number x and returns $4x$.

```
>>> quadruple = funsquare(__<BLANK 1>__)
>>> quadruple(3)  # 4 * 3
12
```

BLANK 1

lambda x: 2 * x

`quadruple` is the result of doubling a number twice, so we should fill blank 1 with a lambda function that doubles its input.

(c) (1.0 pt) Fill in the blank to assign `funfourth` to a function that takes a one-argument function $f(x)$ and returns a one-argument function $f^4(x)$. **You may not use a lambda expression.**

```
>>> funfourth = __<BLANK 2>__
>>> funfourth(triple)(10)  # 10 * 3 * 3 * 3 * 3 (triple is defined near the top of the page)
810
```

BLANK 2

funsquare(funsquare)

$f^4(x)$ is $f(f(f(f(x))))$, which is also $f^2(f^2(x))$. To apply f^2 twice, we call `funsquare(funsquare)`, which returns a function that takes `f` and calls `funsquare(funsquare(f))`.

5. (0.0 points) A+ Question 1: Busy Beaver

This A+ question is not worth any points. It can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first!

To receive an A+ in CS 61A, you need at least 300 points in the course overall and must solve 2 A+ questions. To receive credit for this A+ question, you must answer all parts of it entirely correctly.

Introduced by Tibor Radó in 1962, the *Busy Beaver Game* aims to find the terminating program of a given length that produces the most possible output. Here, we'll explore one approach to generating a lot of output.

Implement `b`, which takes a zero-argument function `f` that returns `None`. The `b` function calls its argument `f` 64 times and returns `None`.

- You may not use any numbers or strings or any expressions that evaluate to numbers or strings.
- You may not import any functions, but may use built-in functions that do not need to be imported.
- You may not use lists, tuples, or dictionaries.
- Your answer for blank 2 must be no longer than 50 characters, ignoring whitespace.

```
def b(f):
    """Evaluate f() 64 times.

    >>> a = lambda: print('A', end='')
    >>> b(a)
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    """
    (lambda g: g(g(g(g(g(g(g(__<BLANK 1>__)))))))(__<BLANK 2>__)__<BLANK 3>__
```

(a) (0.0 pt) `BLANK 1`

- (●) `f`
- ( ) `g`
- ( ) `f()`
- ( ) `g()`

(b) (0.0 pt) `BLANK 2` (**Your answer must not be longer than 50 characters, ignoring whitespace.**)

lambda f: lambda: f() or f()

(c) (0.0 pt) `BLANK 3`

- ● `()`
- ○ `(f)`
- ○ `(g)`

To solve this, we note that $64 = 2^6$, and that there are exactly 6 `g`s in the lambda. Thus, our strategy to call `f` 64 times is as follows:

- Blank 2 should be a function that receives as input a function `h` and returns a new function that calls `h` twice. Since `f` is guaranteed to return None, we can use `h() or h()` to call `h` twice (and return None).
- The first lambda will take in Blank 2 as input `g`, and run g^6 on some input. Since each layer of Blank 2 doubles the number of function calls, g^6 will call its input function $2^6 = 64$ times. This suggests that we should set Blank 1 to `f`.
- Therefore, `(lambda g: g(g(g(g(g(g(__<BLANK 1>__)))))))(__<BLANK 2>__)` creates a function that calls `f()` 64 times. We still need to call it, so Blank 3 should call the newly created function. This can be done by setting Blank 3 to `()`.