CS 61A Fall 2025

Structure and Interpretation of Computer Programs

MIDTERM 1

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.

You can complete 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 Do(ot)?

Assume the code below has been executed, and no errors occurred.

```
about = 6-7
about, face = 6, about * 7

def make_something(f):
    if f(about):
        f = lambda k: about
    def something(about):
        return print(f(about) or print(about))
    return something

f = lambda x: 10 * x
```

Example Question: What would be displayed by evaluating `print(5)`? **Answer**: 5

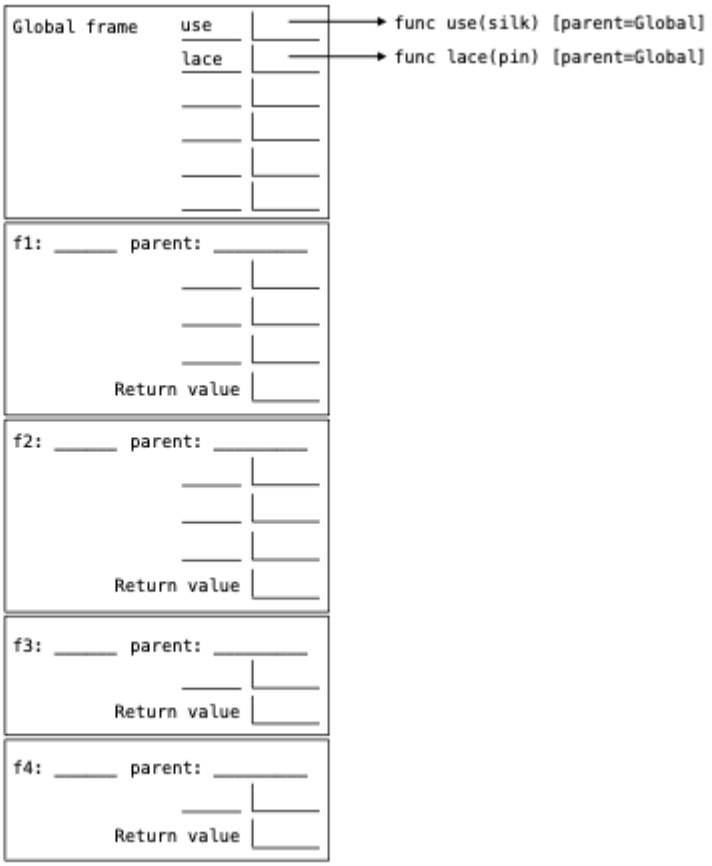
(a) (2.0 pt) What would be displayed by evaluating `print(print(face))`

(b) (4.0 pt) What would be displayed by evaluating `make_something(print)(8)`

2. (5.0 points) Silksong

Complete the environment diagram below to answer the questons about the calls to `print`. Only the questions will be scored, not the diagram. Each call to `print` is evaluated once, and there are no errors caused by running this code.

```
 1: def use(silk):
 2:     a = 2
 3:     print(lace(silk), a)  # (a)
 4:
 5: def lace(pin):
 6:     a = print(pin())      # (b)
 7:     return a
 8:
 9: a = 1
10: thread = lambda x: a + x
11: a = 3
12: needle = lambda: thread(a * 10)
13:
14: use(needle)
15: print(a)                  # (c)
```



(a) (2.0 pt) What is displayed by the call to `print` on **line 3**?

(b) (2.0 pt) What is displayed by the call to `print` on **line 6**?

- ◯ 21
- ◯ 22
- ◯ 23
- ◯ 31
- ◯ 32
- ◯ 33

(c) (1.0 pt) What is displayed by the call to `print` on **line 15**?

◯ 2

◯ 3

◯ None

3. (8.0 points) Saja Boys

Definition. A positive integer is *patterned* if every odd digit has a larger even digit somewhere before it. For example, 4123827 is patterned because the odd digit 7 has 8 (which is even and larger than 7) before it, and the 3 and 1 both have 4 before them.

Implement `patterned`, which takes a positive integer `n`. It returns `True` if `n` is patterned and `False` otherwise. Your implementation should iterate through the digits of `n` just once. The name `odd` in the implementation should always be assigned to an integer from 0 to 9 (inclusive).

```
def patterned(n):
    """Return whether every odd digit of n has a larger even digit somewhere before it.

    >>> patterned(4123827)  # 8 is before 7; 4 is before 1 and 3
    True
    >>> patterned(4412123384137)
    True
    >>> patterned(2468)     # No odd digits
    True

    >>> patterned(1)        # No even digits
    False
    >>> patterned(8192)     # 9 does not have a larger even digit before it (or anywhere)
    False
    >>> patterned(238)      # 3 does not have a larger even digit before it (8 comes after)
    False
    >>> patterned(3888)     # 3 does not have a larger even digit before it (8 comes after)
    False
    >>> patterned(4321587)  # 5 does not have a larger even digit before it (8 comes after)
    False
    """

    odd = 0    # Hint: use the name odd to keep track of a digit you'll need later

    while n:

        n, last = n // 10, n % 10

        if last % 2 == 1:

            -------
              (a)

        elif _______:
               (b)

            -------
              (c)

    return _______
             (d)
```

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

- ◯ `return False`
- ◯ `return max(n) > last`
- ◯ `return n % 10 > last`
- ◯ `return n % 2 == 0 and n % 10 > last`
- ◯ `odd = 0`
- ◯ `odd = last`
- ◯ `odd = n % 10`
- ◯ `odd = max(n % 10, last)`
- ◯ `odd = max(odd, last)`

(b) **(2.0 pt)** Fill in blank (b). You may not use `str` or `len` or `[` or `]`

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

- ◯ `return False`
- ◯ `return max(n) > last`
- ◯ `return n % 10 > last`
- ◯ `return n % 2 == 0 and n % 10 > last`
- ◯ `odd = 0`
- ◯ `odd = last`
- ◯ `odd = n % 10`
- ◯ `odd = max(n % 10, last)`
- ◯ `odd = max(odd, last)`

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

- ◯ `True`
- ◯ `False`
- ◯ `odd == 0`
- ◯ `odd > 0`
- ◯ `last == 0`
- ◯ `last > 0`
- ◯ `last % 2 == 0`
- ◯ `last % 2 == 1`

4. (16.0 points) We're Going Up, Up, Up

Definitions. An *up-sequence* is a sequence of positive integers in which each term is larger than the last. The *next function* `f` for an up-sequence takes a non-negative integer `t` and encodes the sequence as follows: If `t` is any term of the sequence except the last one, then `f(t)` returns the next term in the sequence after `t`. If `t` is the last term of the sequence or not a term in the sequence, `f(t)` returns 0. Finally, when `t` is 0, then `f(0)` returns the first term of the sequence. For example, `fifty_evens` below is a next function for the sequence 2, 4, 6, 8, . . . , 98, 100.

```
def fifty_evens(t):
    "The next function for the finite sequence of the first 50 positive even numbers."
    if t % 2 == 0 and t < 100:
        return t + 2
    return 0
```

(a) (6.0 points)

Implement `sum_sequence`, which takes a next function `f` for a finite up-sequence. It returns the sum of the terms of that sequence.

```
def sum_sequence(f):
    """Return the sum of terms in a finite sequence.

    >>> sum_sequence(fifty_evens)    # 2 + 4 + 6 + 8 + ... + 98 + 100 = 2550
    2550
    """
    t = _______
          (a)

    total = 0
    while t:
        t, total = _______ , _______
                     (b)       (c)
    return total
```

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

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

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

- ◯ `total + 1`
- ◯ `total + t`
- ◯ `total + f`
- ◯ `total + fifty_evens`
- ◯ `total + f(t)`
- ◯ `total + fifty_evens(t)`

(b) (6.0 points)

Here is the next function for the **infinite** up-sequence of positive perfect squares: 1, 4, 9, 16, 25, ...

```
def next_square(t):
    """Compute the next perfect square.

    >>> next_square(0)
    1
    >>> next_square(16)
    25
    >>> next_square(17)  # Not a perfect square
    0
    """
    sqrt_t = t ** 0.5                    # The square root of t
    if sqrt_t == round(sqrt_t):          # Make sure t was a perfect square
        return (round(sqrt_t) + 1) ** 2  # Return the next perfect square
    return 0
```

Implement `cap`, which takes a next function `f` for an **infinite** up-sequence and a positive integer `n`. It returns a next function for the finite up-sequence containing the terms of `f` that are less than or equal to `n`. If `n` is a term of the up-sequence for `f`, then it is included in the result's sequence.

```
def cap(f, n):
    """Return the next function for the up-sequence for f up to (and possibly including) n.

    >>> squares_up_to_25 = cap(next_square, 30)  # 30 is not in the next_square sequence
    >>> squares_up_to_25(4)
    9
    >>> squares_up_to_25(16)
    25
    >>> squares_up_to_25(25)
    0
    >>> squares_up_to_25(17)                     # 17 is not in the next_square sequence
    0
    >>> cap(next_square, 81)(64)                 # 81 is in the next_square sequence
    81
    """
    def capped(t):

        if _______:
             (d)


            _______
              (e)

        else:

            return _______
                     (f)

    return capped
```

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

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

- ◯ `t = 0`
- ◯ `n += 1`
- ◯ `n -= 1`
- ◯ `t += 1`
- ◯ `t -= 1`
- ◯ `t = f(t)`
- ◯ `t = next_square(t)`
- ◯ `return n`
- ◯ `return t`
- ◯ `return 0`

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

- ◯ `n`
- ◯ `t`
- ◯ `f`
- ◯ `f(t)`
- ◯ `next_square`
- ◯ `next_square(t)`

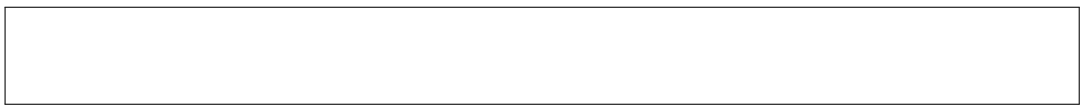
(c) (4.0 points)

Definition. The *previous function* `g` for an up-sequence takes a non-negative integer `t`. If `t` is any other term than the first, `g(t)` is the previous term. If `t` is the first term or not a term at all, `g(t)` returns 0. `g(0)` is the last term.

Implement `reverse`, which takes a next function `f` for a finite up-sequence. It returns the previous function for that same sequence. You may call `max_term`, which is described below, but you don't need to implement `max_term`.

```
def max_term(f):
    """Returns the largest term in the finite up-sequence for next function f.

    >>> max_term(cap(next_square, 20))  # 16 is the largest square less than or equal to 20.
    16
    """
    # Implementation omitted, but you can assume that the function is implemented correctly.

def reverse(f):
    """Return the previous function for the up-sequence encoded by next function f.

    >>> rev_squares = reverse(cap(next_square, 30))  # Goes in reverse through 1, 4, 9, 16, 25
    >>> print(rev_squares(0), rev_squares(25), rev_squares(16), rev_squares(9), rev_squares(4))
    25 16 9 4 1
    >>> rev_squares(1)   # 1 is the first term
    0
    >>> rev_squares(10)  # 10 is not in the sequence
    0
    """

    def previous(t):

        if t == 0:

            return _______
                     (g)
        x = t - 1

        while x > 0 and _______:
                           (h)

            x = x - 1

        return x

    return previous
```

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



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

(d) (0.0 points)

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!

Implement `sum_below`, which takes a next function `f` for an infinite up-sequence and a positive integer `n` that is a term of the sequence. It returns the sum of the terms of the sequence for `f` that are less than (but not including) `n`.

You may **not** use `cap` or `max_term` or `reverse`.

You may **not** use `if` or `else` or `[` or `]`.

```
def sum_below(f, n):
    """Return the sum of the terms of the sequence for f that are below n,
    where n is a term in the sequence for f.

    >>> sum_below(next_square, 25)  # 1 + 4 + 9 + 16
    30
    """
    assert f(n), 'n is not a term of the up-sequence for f'
    return sum_sequence(_______)
                        (i)
```

i. (0.0 pt) Fill in blank (i).

No more questions.