CS 61A Structure and Interpretation of Computer Programs
Spring 2020

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 hand-written 8.5" × 11" crib sheet of your own creation and the official CS 61A midterm 1 study guide.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

Last name	
First name	
Student ID number	
CalCentral email (`_@berkeley.edu`)	
TA	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

POLICIES & CLARIFICATIONS

- 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 `min`, `max`, `pow`, `len`, and `abs`.
- You may not use lists, dictionaries, tuples, sets, or the `:=` operator. These features have not been covered.
- You **may not** use example functions defined on your study guide unless a problem clearly states you can.
- For fill-in-the-blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.

1. (8 points) What Would Python Display *(All are in Scope: WWPD, Higher-Order Functions, Lambda Expressions)*

For each of the expressions in the table 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 first two rows have been provided as examples.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

Assume that you have first started `python3` and executed the statements on the left.

```
def tik(tok):
    tik = lambda: tok + 1
    while tok - tik() < 5:
        tok = tik() + 1
        if tok > 100:
            return tik()
    return tok
```

```
snap = lambda chat: lambda: snap(chat)
snap, chat = print, snap(2020)
```

```
def q(q):
    if print(q, q):
        print(q + 1)
    if q:
        q = q + q
    if q > 0:
        return q
    print(q + 2)
```

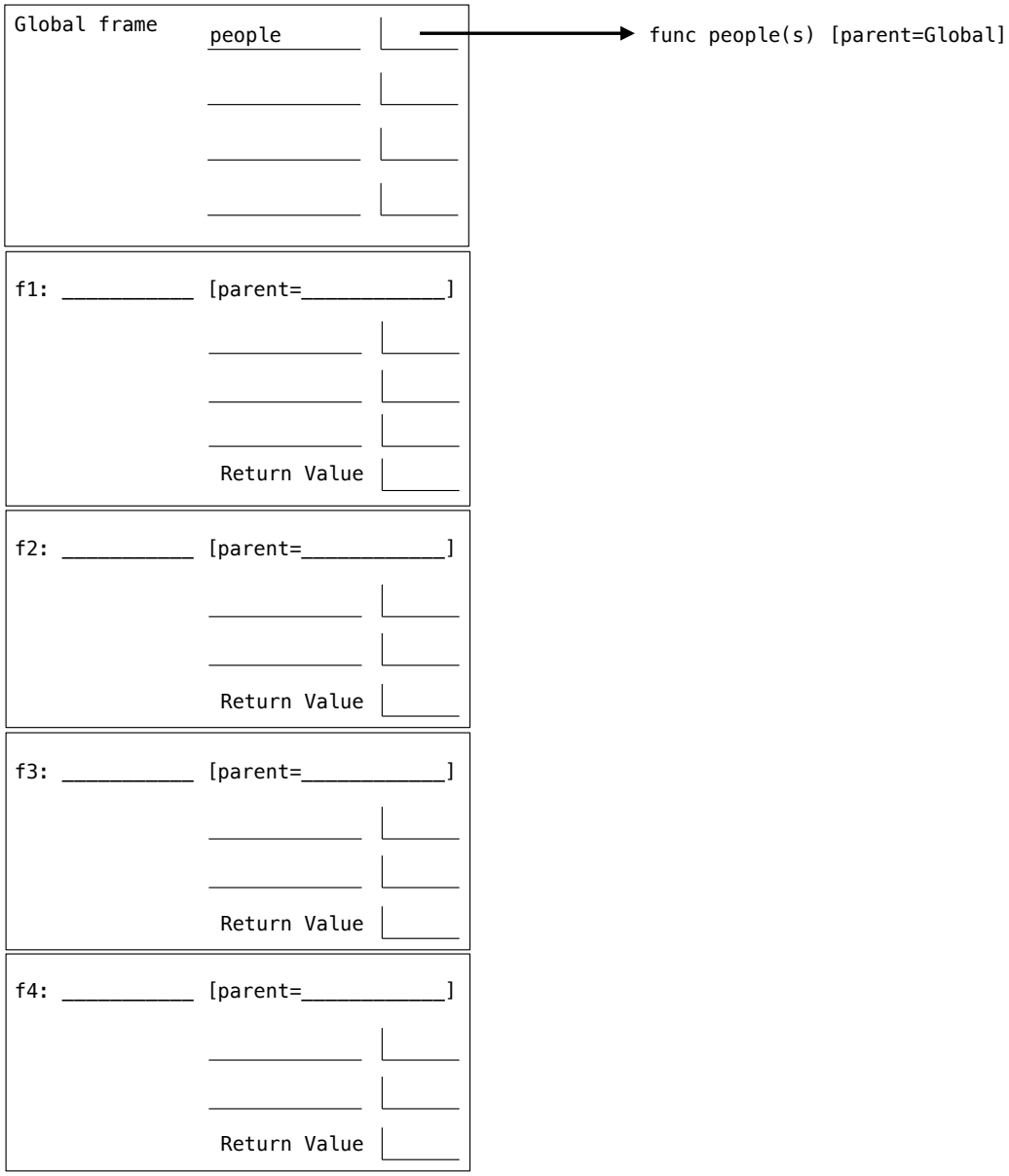
	Expression	Interactive Output
	`pow(10, 2)`	100
	`print(4, 5) + 1`	4 5 Error
(2 pt)	`print(print)(2020)`	
(2 pt)	`tik(50)`	
(1 pt)	`chat(2020)`	
(1 pt)	`chat()`	
(2 pt)	`q(20)`	

2. (8 points) People's Park *(All are in Scope: Environment Diagrams, Higher-Order Functions, Lambda Expressions)*

Fill in the environment diagram that results from executing the code on the right until the entire program is finished, an error occurs, or all frames are filled.
You may not need to use all of the spaces or frames.
Do not include frames for calls to built-in functions.
A complete answer will:

- Add all missing names and parents to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.
- Cross out or erase all arrows and values that are not part of the final diagram.

```
1  def people(s):
2      unit = 2
3      park = s(unit, t) + 1
4      s = (lambda t: park)(s)
5      return lambda: abs(unit)
6
7  unit, t = 1, 3
8  def park(t, park):
9      return unit - 2
10 unit = people(park)
```


Global frame
people
func people(s) [parent=Global]
f1: _____ [parent=_____]
Return Value
f2: _____ [parent=_____]
Return Value
f3: _____ [parent=_____]
Return Value
f4: _____ [parent=_____]
Return Value

3. (16 points) Digit Fidget

(a) (6 pt) *(All are in Scope: Control)* Implement `same_digits`, which takes two positive integers. It returns whether they both become the same number after replacing each sequence of a digit repeated consecutively with only one of that digit. For example, in 12222321, the sequence 2222 would be replaced by only 2, leaving 12321.

Restriction: You may only write combinations of the following in the blanks:
`a`, `b`, `end`, `10`, `%`, `if`, `while`, `and`, `or`, `==`, `!=`, `True`, `False`, and `return`. (**No division allowed!**)

```
def same_digits(a, b):
    """Return whether a and b become the same number after removing adjacent repeats.

    >>> same_digits(2002200, 2202000) # Ignoring repeats, both are 2020
    True
    >>> same_digits(21, 12)           # Digits must appear in the same order
    False
    >>> same_digits(12, 2212)         # 12 and 212 are not the same
    False
    >>> same_digits(2020, 20)         # 2020 and 20 are not the same
    False
    """
    assert a > 0 and b > 0
    while a and b:

        if ____________________________________________________________________________:

            end = a % 10

            ____________________________________________________________________________:

                a = a // 10

            ____________________________________________________________________________:

                b = b // 10

        else:

            ____________________________________________________________________________

    ________________________________________________________________________________
```

(b) (3 pt) *(All are in Scope: Higher-Order Functions)* Implement `no_repeats`, which takes a positive integer `a` and returns the smallest positive integer `b` for which `same_digits(a, b)` returns `True`. Assume `same_digits` is implemented correctly. Watch out for the `assert` statement in the implementation of `same_digits`! You may **not** call `set` or `str`.

```
def no_repeats(a):
    """Remove repeated adjacent digits from a.

    >>> no_repeats(22000200)
    2020
    """
    return search(_____________________________________________________ , _______________)

def search(f, x):
    while not f(x):
        x += 1
    return x
```

(c) (4 pt) *(All are in Scope: Control)* Implement `unique_largest`, which takes a positive integer `n`. It returns whether the largest digit in `n` appears only once in `n`. You may assign values to multiple names in an assignment statement.

You may **not** write call expressions or parentheses. You may **not** write `lambda`, `if`, `max`, `set`, `and`, or `or`.

```
def unique_largest(n):
    """Return whether the largest digit in n appears only once.

    >>> unique_largest(132123)  # 3 is largest and appears twice
    False
    >>> unique_largest(1321523) # 5 is largest and appears only once
    True
    >>> unique_largest(5)
    True
    """
    assert n > 0
    top = 0
    while n:

        n, d = n // 10, n % 10


        if ____________________________________________________________________________:


            ______________________________ = ___________________________________________

        elif d == top:

            unique = ____________________________________________________________________

    return unique
```

(d) (3 pt) *(All are in Scope: Higher-Order Functions)* Implement `transitive`, which takes a two-argument function `p` that returns `True` or `False`. The `transitive` function returns whether it is the case that for every three digits `a`, `b`, `c` for which `p(a, b)` and `p(b, c)` both return `True`, `p(a, c)` also returns `True`. A digit is an integer between 0 and 9, inclusive.

You may **not** write `str`, `[`, or `]`.

```
def transitive(p):
    """Return whether p is transitive over non-negative single digit integers.

    >>> transitive(lambda x, y: x < y)          # if a < b and b < c, then a < c
    True
    >>> transitive(lambda x, y: abs(x-y) == 1)  # E.g., p(3, 4) and p(4, 5), but not p(3, 5)
    False
    """
    abc = 0
    while abc < 1000:

        a, b, c = abc // 100, ________________________________________________, abc % 10


        if p(a, b) ____________________________________________________________________:

            return False
        abc = abc + 1
    return True
```

4. (8 points) Composition

(a) (4 pt) *(All are in Scope: Lambda Expressions, Higher-Order Functions)* Implement `compose`, which takes a positive integer `n`. It returns a function that, when called repeatedly on n one-argument functions $f_1, f_2, \ldots, f_n$, returns a one-argument function of `x` that returns $f_1(f_2(\ldots f_n(x)\ldots))$. You may **not** call the `compose1` function from the Midterm 1 Study Guide.

```
def compose(n):
    """Return a function that, when called n times repeatedly on unary
    functions f1, f2, ..., fn, returns a function g(x) equivalent to
    f1(f2( ... fn(x) ... )).

    >>> add1 = lambda y: y + 1
    >>> compose(3)(abs)(add1)(add1)(-4)  # abs(add1(add1(-4)))
    2
    >>> compose(3)(add1)(add1)(abs)(-4)  # add1(add1(abs(-4)))
    6
    >>> compose(1)(abs)(-4)              # abs(-4)
    4
    """
    assert n > 0


    if n == 1:


        return ____________________________________________________________________________


    def call(f):


        def on(g):


            return ____________________________(_____________________________________)


        return on


    return call
```

(b) (4 pt) *(All are in Scope: Lambda Expressions, Higher-Order Functions)* Complete the final expression below with **only integers and names** so it evaluates to 2020.

```
from operator import add

c = lambda f: lambda x: lambda y: f(x, y)
twice = lambda z: 2 * z


compose(__________)(twice)(__________(__________)(10))(__________(pow)(10))(__________)
```