CS 61A Structure and Interpretation of Computer Programs
Summer 2019

Final

INSTRUCTIONS

- You have 3 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except three hand-written 8.5" × 11" crib sheet of your own creation and the official CS 61A final 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	
Exam Room	
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`, `abs`, `sum`, `next`, `iter`, `list`, `tuple`, `map`, `filter`, `zip`, `all`, and `any`.
- You **may not** use example functions defined on your study guides 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.
- You may use the `Tree` and `Link` classes defined on Page 4 (left column) of the Study Guide.

1. (12 points) Calling Card

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. The first row is completed for you.

- If an error occurs, write **Error**, but include all output displayed before the error.
- To display a function value, write **Function**.
- If an expression would take forever to evaluate, write **Forever**.

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

Assume that you have started `python3` and executed the code shown on the left first, then you evaluate each expression on the right in the order shown. Expressions evaluated by the interpreter have a cumulative effect.

```
class Card:
    num = 0

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
        Card.num += 1

    def __eq__(self, card):
        print(self.num)
        return (self.suit == card.suit) \
               and (self.rank == card.rank)

class Deck(Card):
    suits = ['H', 'C']
    ranks = ['A'] + list(range(2, 3))

    def __init__(self, cards=[]):
        self.cards = cards
        if not cards:
            for suit in self.suits:
                for rank in self.ranks:
                    card = Card(suit, rank)
                    self.cards.append(card)

    def get_cards(self):
        i = 0
        while i < 2:
            yield self.suits[i]
            i += 1
        yield self.ranks

deck = Deck([])
Card.num += 1
cards = deck.get_cards()
```

Expression	**Output**
`print(None)`	None
`Card('H', 'A') is Card('H', 'A')`	
`Card.num != deck.num`	

`Deck.__init__ = Card.__init__`
`Deck(Deck.suits[0],`
`     Deck.ranks[1]).rank`	

`deck == deck`	
`next(cards)`	
`list(cards)`	

2. (10 points) Strangest Things

Fill in the environment diagram that results from executing the code on the left 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.* A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.
- **Use box-and-pointer diagrams for lists and tuples.**

```
def stranger(strings):
    mike = lambda s: dog.append(dog[0])
    def strings(eleven):
        nonlocal strings
        strings = mike
        return strings
    return strings

def hopper():
    i = 0
    dog[i] = stranger(stranger)("mike")

dog = ['d', 'e', 'm', 'o']
strings = dog
stranger = stranger(dog)
hopper()
```

Global frame

stranger → func stranger(strings) [parent=Global]

hopper → func hopper() [parent=Global]

dog

f1: ________ [parent: ________]

Return Value

f2: ________ [parent: ________]

________ 0

Return Value

f3: ________ [parent: ________]

Return Value

f4: ________ [parent: ________]

Return Value None

3. (14 points) One More Time

Definition. An (n)-*repeater* for a single-argument function `f` takes a single argument `x`, calls `f(x)` n times, then returns an $(n+1)$-repeater for `f`.

(a) (6 pt) Implement `repeater`, which takes a single-argument function `f` and a positive integer n. It returns an (n)-repeater for `f`. Also implement the helper function `repeat`.

```
def repeater(f, n):
    """Return an (n)-repeater for f.

    >>> r = repeater(print, 2)
    >>> s = r('CS')
    CS
    CS
    >>> t = s('CS')
    CS
    CS
    CS
    """


    def g(x):


        ______________________________________________________________________


        return _______________________________________________________________


    return ___________________________________________________________________


def repeat(f, x, n):
    """Call f(x) n times.

    >>> repeat(print, 'Hello', 3)
    Hello
    Hello
    Hello
    """


    if _______________________________________________________________________:


        _____________________________________________________________________


        _____________________________________________________________________
```

(b) **(6 pt)** Implement `compound`, which takes a single-argument function `f` and returns a single-argument function `g`. When `g` is called for the nth time, it returns the result of calling `f` repeatedly n times. That is, the first call `g(x)` returns `f(x)`, the second call `g(y)` returns `f(f(y))`, and the third call `g(z)` returns `f(f(f(z)))`. Do **not** call `repeat` or `repeater` in your implementation.

```
def compound(f):
    """Return a function that, when called the nth time, applies f repeatedly n times.

    >>> double = lambda y: 2 * y
    >>> doubler = compound(double)
    >>> doubler(3)                 # 1st call to doubler; double 3 one time
    6
    >>> doubler(5)                 # 2nd call to doubler; double 5 two times
    20
    >>> doubler(7)                 # 3rd call to doubler; double 7 three times
    56
    """

    h = ______________________________________________________________________________

    def g(x):

        ______________________________________________________________________________

        h = __________________________________________________________________________

        return h(x)

    return g
```

(c) **(2 pt)** Write the values bound to `b` and `c` that result from executing the code below, assuming `compound` is implemented correctly.

```
increment = lambda x: x + 1
h = compound(compound(increment))
a, b, c = h(3), h(3), h(3)
```

```
b: _________________        c: _________________
```

4. (14 points) Combo Nation

Definition. A *combo* of a non-negative integer n is the result of adding or multiplying the digits of n from left to right, starting with 0. For $n = 357$, combos include $15 = (((0 + 3) + 5) + 7)$, $35 = (((0 * 3) + 5) * 7)$, and $0 = (((0 * 3) * 5) * 7)$, as well as 0, 7, 12, 22, 56, and 105. But $36 = ((0 + 3) * (5 + 7))$ is not a combo of 357.

(a) (6 pt) Implement `is_combo`, which takes non-negative integers n and k. It returns whether k is a combo of n. You may assume that 0 is not one of the digits of n.

```
def is_combo(n, k):
    """Is k a combo of n?

    >>> [k for k in range(1000) if is_combo(357, k)]
    [0, 7, 12, 15, 22, 35, 56, 105]
    """
    assert n >= 0 and k >= 0


    if ______________________________________________________________________________:


        return True


    if ______________________________________________________________________________:


        return False


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


    added = _______________________ and is_combo(______________, ____________________)


    multiplied = _______________________ and is_combo(_____________, _________________)


    return added or multiplied
```

(b) **(4 pt)** Implement `apply_tree`, which takes in two trees. The labels of the first tree are all functions. `apply_tree` should mutate the the second tree such that each label is the result of applying each function in the first tree to the corresponding node in the second tree.

You may assume the two trees have the same shape (that is, each node has the same number of children).

```
def apply_tree(fn_tree, val_tree):
    """ Mutates val_tree by applying each function stored in fn_tree
    to the corresponding labels in val_tree

    >>> double = lambda x: x*2
    >>> square = lambda x: x**2
    >>> identity = lambda x: x
    >>> t1 = Tree(double, [Tree(square), Tree(identity)])
    >>> t2 = Tree(6, [Tree(2), Tree(10)])
    >>> apply_tree(t1, t2)
    >>> t2
    Tree(12, [Tree(4), Tree(10)])
    """

    ___________________________________________________________________________

    for __________________________ in _________________________________________:

        ______________________________________________________________________
```

(c) **(4 pt)** Implement `make_checker_tree` which takes in a tree, `t` containing digits as its labels and returns a tree with functions as labels (a function tree). When applied to another tree, the function tree should mutate it so each label is `True` if the label is a combo of the number formed by concatenating the labels from the root to the corresponding node of `t`. You may use `is_combo` in your solution.

The default argument for `make_checker_tree` is part of the solution, but will not be present in the initial call.

```
def make_checker_tree(t, so_far=0):
    """ Returns a function tree that, when applied to another tree, will mutate its labels to be
    True if the label is a combination of the path in t from the root to its corresponding node.

    >>> t1 = Tree(5, [Tree(2), Tree(1)])
    >>> fn_tree = make_checker_tree(t1)
    >>> t2 = Tree(5, [Tree(10), Tree(7)])
    >>> apply_tree(fn_tree, t2) #5 is a combo of 5, 10 is a combo of 52, 7 isn't a combo of 51
    >>> t2
    Tree(True, [Tree(True), Tree(False)])
    """

    new_path = ________________________________________________________

    branches = ________________________________________________________

    fn = ______________________________________________________________

    return Tree(fn, branches)
```

5. (6 points) Back of the line

Implement a function `bouncer` that takes in a linked list and an index i and moves the value at index i of the link to the end. You should mutate the input link.

You should implement `swapper` to help with your implementation.

You may assume that i is non-negative and less than the length of the linked list.

Use the following implementation of `Link`:

```
class Link:
    """A linked list."""
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(self.first, rest_str)
```

You may not use any methods that are not given in the above implemenetation in your solution.

```
def bouncer(link, k):
    """
    >>> lnk = Link(5, Link(2, Link(7, Link(9))))
    >>> bouncer(lnk, 0)
    >>> lnk
    Link(2, Link(7, Link(9, Link(5))))
    >>> bouncer(lnk, 2)
    >>> lnk
    Link(2, Link(7, Link(5, Link(9))))
    """

    if ________________________________________________________________________:

        ________________________________________________________________________

    else:

        ________________________________________________________________________

def swapper(link):

    if ________________________________________________________________________:

        return

    _________________, _________________ = _________________, _________________

    ________________________________________________________________________
```

6. (2 points) Before we get to Scheme, some cons-ceptual questions

There are three problems here, which cover topics from the two special topics lectures. Each problem is worth 1 point, but you can only earn a maximum of 2 points on this problem, so you only need to know two answers.

Please make sure to fill in the bubble completely when answering. Each question has only one right answer.

(a) (1 pt) Security

Which of the following was NOT presented a defense against phishing in lecture?

◯ 2-Step Authentication
◯ Cross-Site Request Forgery
◯ U2F Keys
◯ Inspecting the grammar and spelling of a suspicious email

(b) (1 pt) Complexity

As presented in lecture, which of the following best describes the difference between a problem in P and a problem in NP (but not in P)?

◯ It takes a long time to check if a solution is correct for a problem in NP (but not P), and a short time for a problem in P
◯ Some problems in NP are uncomputable, but every problem in P is computable
◯ It takes a long time to come up with a solution for a problem in NP (but not P), and a short time for a problem in P

(c) (1 pt) Computability

Which of the following was discussed as an implication of the halting problem in lecture?

◯ Under no circumstances can we determine if a program will terminate for any valid input
◯ It is impossible to write an antivirus that can always determine if a program will execute malicious code
◯ The problem of determining if an *arbitrary* program will terminate on any input is NP-complete

7. (12 points) Procedure with Caution

Definition. A *sequential procedure* takes a non-negative integer i as an argument and returns the ith element of an infinite sequence. The `i`th element of a sequential procedure `f` is `(f i)`.

A sequential procedure starts at element 0 (so to get the first element for sequential procedure `f`, you'd call `(f 0)`)

Doctests are listed **after** the skeleton code for each question.

(a) (4 pt) Implement `streamify`, which takes a sequential procedure and returns a *stream* containing its elements.

```
(define (streamify f)

  (define (g n)


    ________________________________________________________)


  ________________________________________)

(streamify (lambda (n) 4))             ; An infinite stream of 4 4 4 4 ...
(streamify (lambda (n) (abs (- n 4)))) ; An infinite stream of 4 3 2 1 0 1 2 3 4 ...
```

(b) (5 pt) Implement the Scheme procedure `duplicate` that returns the first duplicate element of a sequential procedure. Assume a duplicate element exists. You may call the `contains?` procedure defined below. Your procedure **must be tail recurisve** in order to receive credit.

```
(define (contains? s v)

  (cond ((null? s) false)

        ((equal? (car s) v) true)

        (else (contains? (cdr s) v))))

(define (duplicate f)

  (define (helper ____________________________________________________________)


    (if ______________________________________________________________________


        ______________________________________________________________________


        (helper ______________________________________________________________)))


  (helper ____________________________________________________________________))

(duplicate (lambda (n) 4))                      ; returns 4, sequence is 4 [4] ...
(duplicate (lambda (n) (abs (- n 4))))          ; returns 1, sequence is 4 3 2 1 0 [1] 2 ...
(duplicate (lambda (n) (remainder (+ n 3) 4))) ; returns 3, sequence is 3 0 1 2 [3] 4 0 ...
```

(c) (3 pt) Implement `slice`, a macro that takes a sequential procedure `f` and a non-negative integer `k` using the syntax `(slice f at k)`. It returns a new sequential procedure whose ith element is element $i + k$ of `f`. You may assume that after we create a `slice` for `f` that `f` will never be reassigned.

```
(define-macro (slice f at k)


  _______________________________________________________)

(define (f x) (+ x 2))     ; f sequence is 2 3 4 5 6 7 ...
(define g (slice f at 3)) ; g sequence is       5 6 7 ...
(g 2)                      ; expect 7
```

8. (10 points) The Big SQL

The `ingredients` table describes the `dish` and `part` for each part of each dish at a restaurant. The `shops` table describes the `food`, `shop`, and `price` for each part of a dish sold at each of two shops. All ingredients (parts) are sold by both shops, and each ingredient will only appear once for each shop. Write your SQL statements so that they would still be correct if table contents changed. You can assume the `shops` table will only ever contain two shops (`A` and `B`).

```
CREATE TABLE ingredients AS
  SELECT "chili" AS dish, "beans" AS part UNION
  SELECT "chili"        , "onions"        UNION
  SELECT "soup"         , "broth"         UNION
  SELECT "soup"         , "onions"        UNION
  SELECT "beans"        , "beans";
```

```
CREATE TABLE shops AS
  SELECT "beans" AS food, "A" AS shop, 2 AS price UNION
  SELECT "beans"        , "B"        , 2 AS price UNION
  SELECT "onions"       , "A"        , 3          UNION
  SELECT "onions"       , "B"        , 2          UNION
  SELECT "broth"        , "A"        , 3          UNION
  SELECT "broth"        , "B"        , 5;
```

(a) (2 pt) Select a two-column table with one row per food that describes the lowest price for each food.

```
SELECT food,  _________________ FROM shops ________________________;
```

beans	2
broth	3
onions	2

(b) (4 pt) Select a two-column table with one row per dish that describes the total cost of each dish if all parts are purchased from shop A.

```
SELECT ________________________ FROM _____________________________

  WHERE __________________________________________________________;
```

beans	2
chili	5
soup	6

(c) (4 pt) In two different ways, select a one-column table of all foods that have a different price at each store.

```
SELECT _________ FROM __________________________________, __________________________________

   WHERE ____________________________________________________________________________________;

SELECT _________ FROM shops GROUP BY _______________________________________________________;
```

onions
broth

9. (0 points) The End

(a) (0 pt) Any feedback for us on how this exam went / your experience in the course?

(b) (0 pt) Use this space to draw a picture, write a note, or otherwise express yourself.