CS 61A Fall 2018 — Structure and Interpretation of Computer Programs

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 midterm 1, midterm 2, and final study guides.
- 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`, `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`, `Link`, and `BTree` classes defined on Page 2 (left column) of the Midterm 2 Study Guide.

1. (12 points) High Quality Air

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**.
- To display an iterator value, write **Iterator**.
- 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.

```
from operator import sub

z = (lambda x: lambda y: 2 * (y-x))(3)

def breath(f, count=1):
    if count > 1:
        print(count)
    count += 1
    return lambda x, y: f(x+1, y)

class Day:
    aqi = 10
    def __init__(self, aqi=0):
        if aqi > self.aqi:
            self.aqi = aqi
        self.n = []
    def mask(self, limit):
        def f(aqi):
            if aqi > limit:
                self.n.append(aqi-limit)
            return self.mask(aqi)
        return f

class Week(Day):
    aqi = 50

m, t = Day(), Week(199)
t.mask(200)(100)(150)(160)
Day.aqi = 140
t.aqi = 160
```

Expression	**Output**
`print(None)`	None
`print(print(None), print)`	
`z(4)`	
`breath(breath(sub))(5, 3)`	
`[Day().aqi, m.aqi]`	
`[Week.aqi, t.aqi]`	
`t.n`	

2. (8 points) Diagram Horror

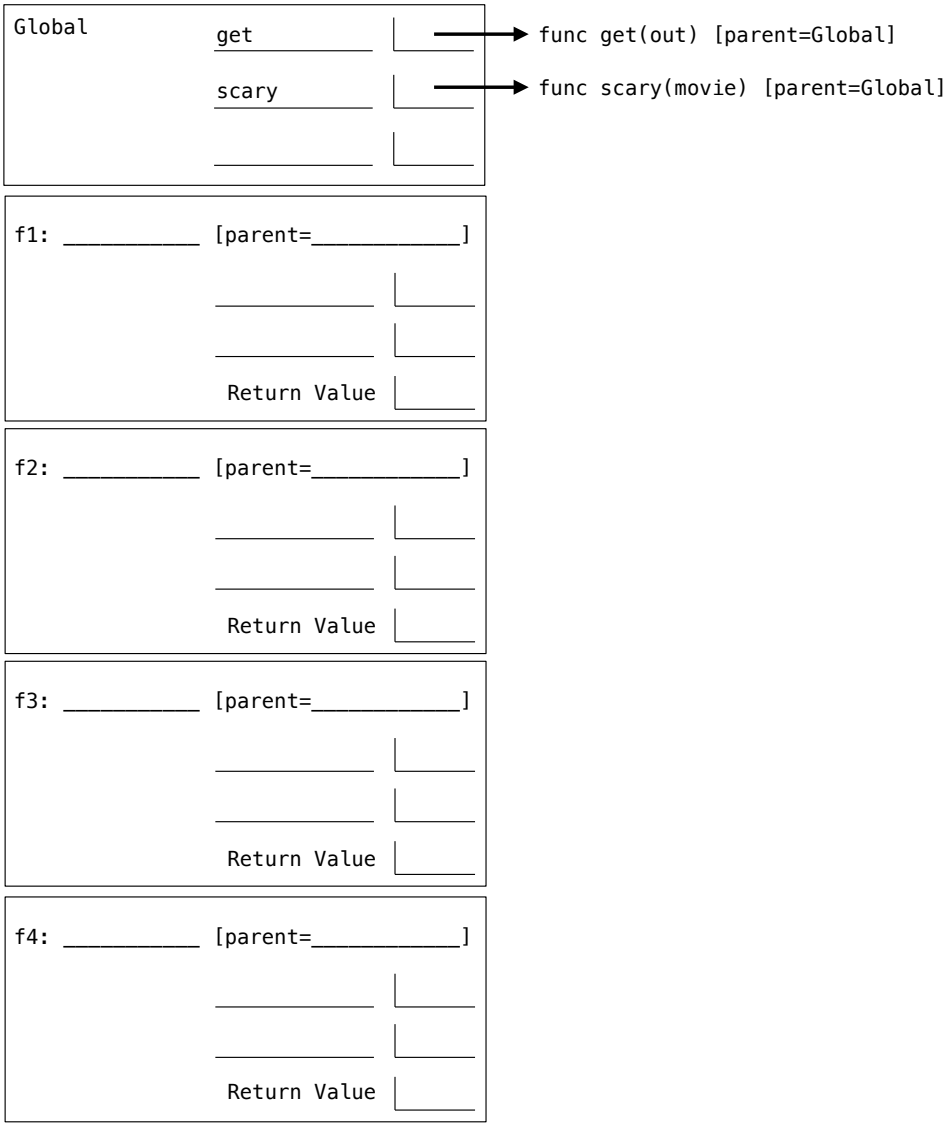
```
def get(out):
    out.pop()
    out = scary(lambda movie: out)
    return lambda: [out]

def scary(movie):
    out.append(movie)
    return movie(5)[:1]

out = [6]
get([7, 8])()
```

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.**



3. (16 points) Gainz

Definition. A sequence is *near increasing* if each element beyond the second is larger than all elements preceding its previous element. That is, element i must be larger than elements $i-2$, $i-3$, $i-4$, etc.

(a) (3 pt) Implement `is_near`, which takes a sequence `s` and returns whether its elements form a near increasing sequence.

```
def is_near(s):
    """Return whether s is a near increasing sequence.

    >>> is_near([]) and is_near([1]) and is_near([1, 2]) and is_near(range(10))
    True
    >>> is_near([4, 2]) and is_near((1, 4, 2, 5)) and is_near((1, 2, 4, 3))
    True
    >>> is_near((3, 2, 1))        # 1 <= 3
    False
    >>> is_near([1, 4, 2, 3, 5])  # 3 <= 4
    False
    >>> is_near([1, 4, 2, 5, 3])  # 3 <= 4
    False
    >>> is_near([1, 2, 4, 2, 5])  # 2 <= 2
    False
    """

    return all([__________ > _____________________ for i in ____________________________])
```

(b) (6 pt) Implement `fast_near`, which takes an iterable value and returns whether its elements form a near increasing sequence. `fast_near` must run in $\Theta(n)$ time and $\Theta(1)$ space (not including the input itself) for an iterable input with n elements. Assume that `s` has a finite number of elements. You may **not** call `is_near`.

```
def fast_near(s):
    """Return whether the elements in iterable s form a near increasing sequence.

    >>> fast_near([2, 5, 3, 6, 6, 7, 7, 9, 8])
    True
    """

    t, s = iter(s), None  # Do not refer to s below this line.

    try:

        largest, last = _________________________________, ______________________________

    except StopIteration:

        return ____________________________________________________________________________

    for x in t:

        if ____________________________________________________________________________:

            return False

        largest, last = _________________________________, ______________________________

    return True
```

Alternative Definition. *(Equivalent to the one on the previous page, but stated in a more useful way for the problem below.)* A sequence is *near increasing* if each element but the last two is smaller than all elements following its subsequent element. That is, element i must be smaller than elements $i+2$, $i+3$, $i+4$, etc.

(c) (6 pt) Implement `near`, which takes a non-negative integer `n` and returns the largest near increasing sequence of digits within `n` as an integer. The arguments `smallest` and `d` are part of the implementation; you must determine their purpose. You may **not** call `is_near` or `fast_near`. You may **not** use any values except integers and booleans (`True` and `False`) in your solution (no lists, strings, etc.).

```
def near(n, smallest=10, d=10):
    """Return the longest sequence of near-increasing digits in n.

    >>> near(123)
    123
    >>> near(153)
    153
    >>> near(1523)
    153
    >>> near(15123)
    1123
    >>> near(11111111)
    11
    >>> near(985357)
    557
    >>> near(14735476)
    143576
    >>> near(812348567)
    1234567
    """
    if n == 0:

        return ____________________________________________________________________________

    no = near(n//10, smallest, d)

    if smallest > ____________________________________________________________________________:

        yes = ____________________________________________________________________________

        return ____________________________________________________________________(yes, no)

    return ____________________________________________________________________________
```

(d) (1 pt) What is the largest possible integer that could ever be returned from the `near` function? **Note**: In general, integers in Python can be arbitrarily large.

_____

4. (11 points) Tree Time

Definition. A *runt node* is a node in a tree whose label is smaller than all of the labels of its siblings. A sibling is another node that shares the same parent. A node with no siblings is a runt node.

(a) (7 pt) Implement `runts`, which takes a `Tree` instance `t` in which *every label is different* and returns a list of the labels of all runt nodes in `t`, in any order. Also implement `apply_to_nodes`, which returns nothing and is part of the implementation. Do **not** mutate any tree. The `Tree` class is on the Midterm 2 Guide.

```
def runts(t):
    """Return a list in any order of the labels of all runt nodes in t.

    >>> sorted(runts(Tree(9, [Tree(3), Tree(4, [Tree(5, [Tree(6)]), Tree(7)]), Tree(2)])))
    [2, 5, 6, 9]
    """
    result = []
    def g(node):

        if ____________________________________________________________________________:


            result.append(_____________________________________________________________)


    apply_to_nodes(_____________________________________________________________________)


    return ____________________________________________________________________________

def apply_to_nodes(f, t):
    """Apply a function f to each node in a Tree instance t."""


    ____________________________________________________________________________________

    for b in t.branches:


        ____________________________________________________________________________
```

(b) (4 pt) Implement `max_label`, which takes a `Tree` `t` and returns its largest label. Do **not** mutate any tree.

```
def max_label(t):
    """Return the largest label in t.

    >>> max_label(Tree(4, [Tree(5), Tree(3, [Tree(6, [Tree(1), Tree(2)])])]))
    6
    """
    def f(node):

        ____________________________________________________________________________


        _______ max(____________, ____________, key=lambda n: _________________________)

    apply_to_nodes(f, t)  # Assume that apply_to_nodes above is implemented correctly.

    return t.label
```

5. (9 points) Run, Program, Run

Implement `runs`, a Scheme procedure that takes a list of integers `s` and returns a list of non-empty lists of integers `t`. Together, the lists in `t` should contain all elements of `s` in order. The first element in each list in `t` must be less than the last element in the previous list, if there is one. The rest of the elements in each list in `t` must be greater than or equal to the previous element.

Also implement and use `next-run` in your solution, which takes a non-empty list of integers `s` and returns a pair of lists: the longest non-decreasing prefix of `s` and the rest of `s`. Use the provided `pair` data abstraction. Your implementation should be correct even if the `pair` implementation were to change.

```
;; Return a list of non-decreasing lists that together contain the elements of s.
;; scm> (runs '(3 4 7 6 6 8 1 2 5 5 4))
;; ((3 4 7) (6 6 8) (1 2 5 5) (4))
;; scm> (runs '(4 3 2 3))
;; ((4) (3) (2 3))
(define (runs s)


    (if (null? s) ____________________________________________________________________


        (let ((p (next-run s)))


            ____________________________________________________________________)))

;; A data abstraction for a pair of a first run and the rest of a list.
(define (pair a b) (lambda (c) (if c a b)))
(define (first p) (p #t))
(define (rest p) (p #f))

;; Return a pair containing the first run in s (a list) and the rest of s (another list).
;; scm> (first (next-run '(4 5 1 3 2)))
;; (4 5)
;; scm> (rest (next-run '(4 5 1 3 2)))
;; (1 3 2)
(define (next-run s)


    (if (or ____________________________________________________________________


            ____________________________________________________________________)


        (pair ____________________________________  ____________________________________)


        (begin


            (define p (next-run (cdr s)))


            (pair ____________________________  ____________________________________))))
```

6. (9 points) Generation Z

(a) (4 pt) Implement `rev`, a generator function that takes a `Link` instance and yields the elements of that linked list in reverse order. The `Link` class appears on Page 2 of the Midterm 2 Study Guide.

```
def rev(s):
    """Yield the elements in Link instance s in reverse order.

    >>> list(rev(Link(1, Link(2, Link(3)))))
    [3, 2, 1]
    >>> next(rev(Link(2, Link(3))))
    3
    """
    if ____________________________________________________________________________________:


        ____________________________________________________________________________________


        yield ____________________________________________________________________________
```

(b) (2 pt) Using the provided `add` procedure, define `not-three`, an infinite stream of all positive integers that are not evenly divisible by 3. The `not-three` stream is increasing and begins with 1, 2, 4, 5, 7, 8, 10, 11, 13.

```
(define (add k s) (cons-stream (+ k (car s)) (add k (cdr-stream s))))


(define not-three ____________________________________________________________________________)
```

(c) (3 pt) Implement `infix`, a Scheme macro that evaluates infix expressions. An infix expression is either a number or a three-element list containing an infix expression, a procedure, and another infix expression. The value of a compound infix expression is the value of its second element applied to the values of its first and third elements. **Note**: The last line begins with a quasiquote. If you cross out the quasiquote and solve the problem without using quasiquote or unquote, you can receive up to 2 out of 3 points (not recommended).

```
;; A macro to evaluate infix expressions.
;; scm> (infix (2 * 3))
;; 6
;; scm> (infix ((1 + 1) * (1 + 2)))
;; 6
;; scm> (infix ((1 + (3 - 2)) * ((2 + 3) + 2)))
;; 14
(define-macro (infix e)

  (if (number? e) e


    `(____________________________________________________________________________)))

(define (cadr x) (car (cdr x)))
(define (caddr x) (car (cdr (cdr x))))
```

7. (10 points) SQL of Course

The `courses` table describes the `course` name, start time hour (`h`) and minute (`m`), and length in minutes (`len`) for different lectures. For example, 61A starts at 13:00 and lasts 50 minutes. The `locations` table describes the course `name` and location (`loc`) of these courses. Assume that each course name appears exactly once in each table. Write your SQL statements so that they would still be correct if the table contents changed.

```
CREATE TABLE courses AS
  SELECT "1" AS course, 14 AS h, 0 AS m, 80 AS len UNION
  SELECT "2"          , 13     , 30    , 80        UNION
  SELECT "8"          , 12     , 30    , 50        UNION
  SELECT "10"         , 12     , 30    , 110       UNION
  SELECT "50AC"       , 13     , 30    , 45        UNION
  SELECT "61A"        , 13     , 0     , 50;
```

```
CREATE TABLE locations AS
  SELECT "1" AS name, "VLSB" AS loc UNION
  SELECT "2"        , "Dwinelle"    UNION
  SELECT "10"       , "VLSB"        UNION
  SELECT "50AC"     , "Wheeler"     UNION
  SELECT "61A"      , "Wheeler";
```

(a) (2 pt) Select a one-column table that contains the course names of all courses that start **before** 13:30.

```
SELECT course FROM courses WHERE _____________________________________________________;
```

61A
8
10

(b) (4 pt) Select a two-column table with one row per location that contains the location, as well as the shortest length in minutes of any lecture held in that location.

```
SELECT loc, ____________________________________________________________________________

  FROM ____________________________________________________________________________

  ____________________________________________________________________________;
```

Dwinelle	80
VLSB	80
Wheeler	45

(c) (4 pt) Select a three-column table where each row describes an earlier course, a later course, and the amount of time in minutes between the end time of the earlier course and the start time of the later course. Only include pairs of courses where the lectures do not overlap in time. *Note*: There are 60 minutes in an hour.

```
SELECT ____________________________________ , ____________________________________

  ____________________________________ AS gap

  FROM ____________________________________________________________________________

  ____________________________________________________________________________;
```

61A	1	10
8	1	40
8	2	10
8	50AC	10

8. (0 points) Draw! *(Optional)* Draw a picture of some function or procedure.