CS 61A Fall 2017

Structure and Interpretation of Computer Programs

FINAL SOLUTIONS

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 sheets of your own creation and the official CS 61A midterm 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.
- Before asking a question, read the announcements on the screen/board. We will not answer your question directly. If we decide to respond, we'll add our response to the screen/board so everyone can see the clarification.
- 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. (10 points) Calling All Values

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 interactive interpreter displays the repr string of the value of a successfully evaluated expression, unless it is `None`. Write "FUNC" to indicate a functional value.

The first two rows have been provided as an example.

Assume that you have started `python3` and executed all the code to the left of the table first.

```
fandv = lambda f, x: [f, f(x)]
```

```
def pv(v):
    print(v)
    return v
dbl = lambda x: 2*x
Idbl = lambda: pv(lambda x: x) or pv(dbl)

def upto(n):
    items = []
    for i in range(n):
        items.append(i)
        yield items
def av(v):
    v.append(-1)
    return v
```

```
def rc(f, n):
    def g(y): return [n, f(y)]
    return rc(g, n // 2) if n>2 else g(n)
```

```
def mx(x):
    x += 3
```

Expression	Interactive Output
`[2, 3]`	[2, 3]
`print((2, 3))`	(2, 3)
`fandv(print, print)`	FUNC [FUNC, None]
`Idbl()(pv(17) and pv(1))`	FUNC 17 1 1
`[ av(x) for x in upto(2)][0]`	[0, -1, 1, -1]
`rc(lambda x: x, 9)`	[2, [4, [9, 2]]]

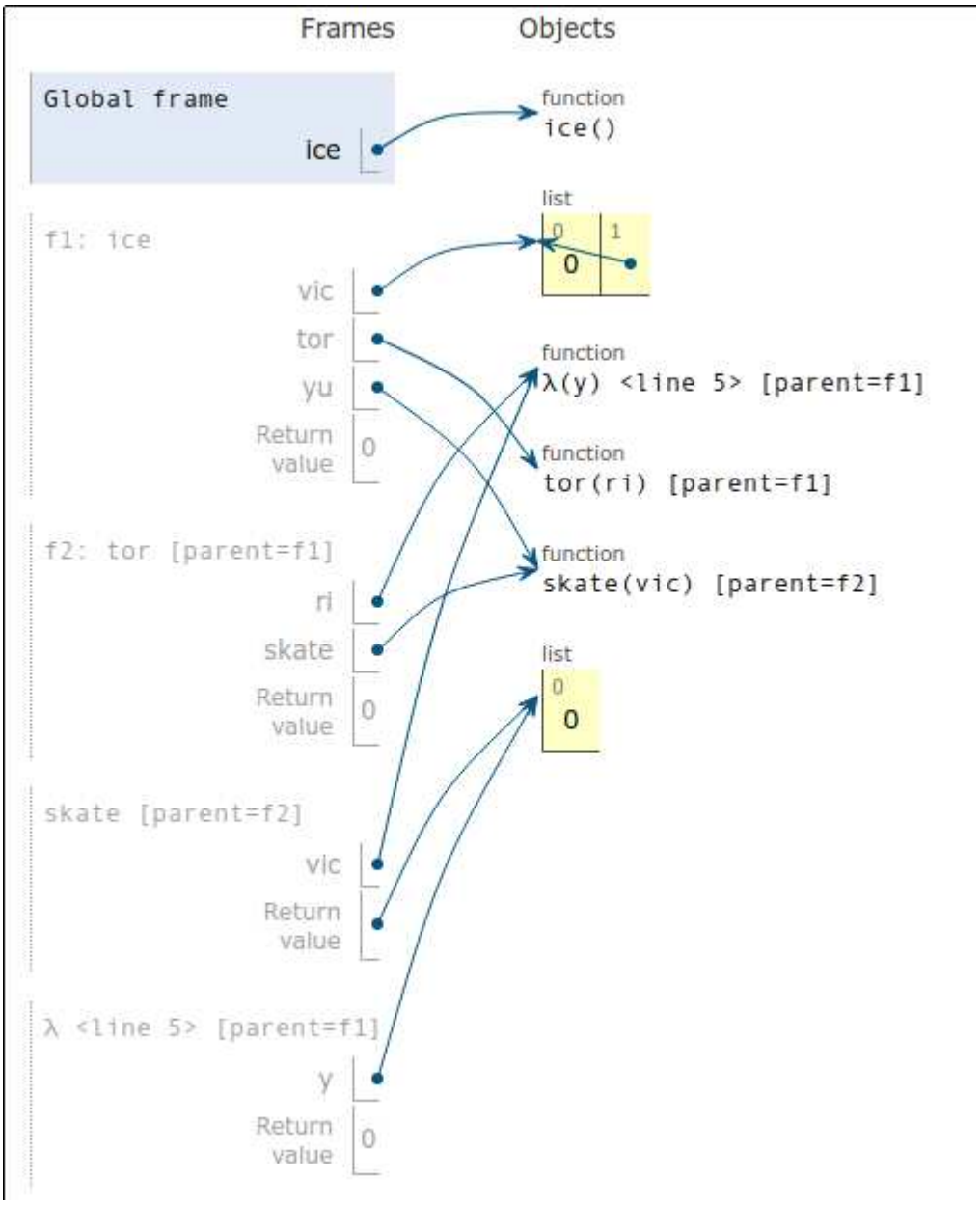
`z=4`
`mx(z)`
`print(z)`
	4

2. (10 points) Environmentally Friendly

Fill in the environment diagram that results from executing the code below 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:

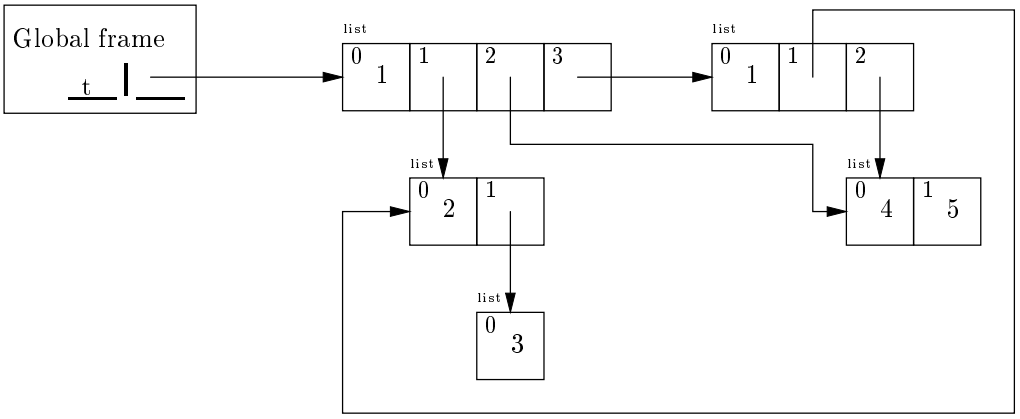
- Add all missing names and parent annotations to frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.
- Use box-and-pointer notation for list values. You do not need to write index numbers or the word "list".



3. (8 points) Get the Point? Fill in the environment diagram that results from executing each block of code below until the entire program is finished or an error occurs. Use box-and-pointer notation for lists. You don't need to write index numbers or the word list. Erase or cross out any boxes or pointers that are not part of a final diagram.

a. (3 pt)

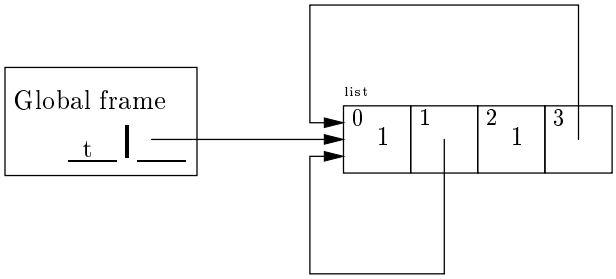
```
t = [1,[2,[3]],[4,5]]
t.append(t[:])
```



---

b. (2 pt)

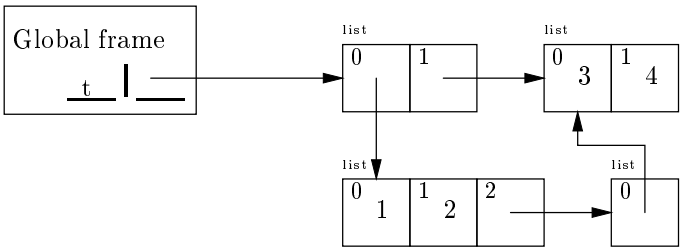
```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```



---

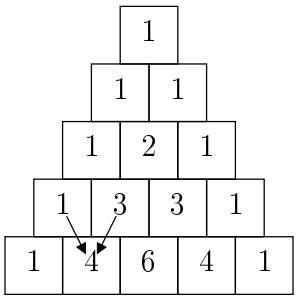
c. (3 pt)

```
t = [[1,2],[3,4]]
t[0].append(t[1:2])
```



4. (14 points) O! Pascal

Pascal's Triangle is perhaps familiar to you from the diagram below, which shows the first five rows.



Every square is the sum of the two squares above it (as illustrated by the arrows showing here the value 4 comes from), unless it doesn't have two squares above it, in which case its value is 1.

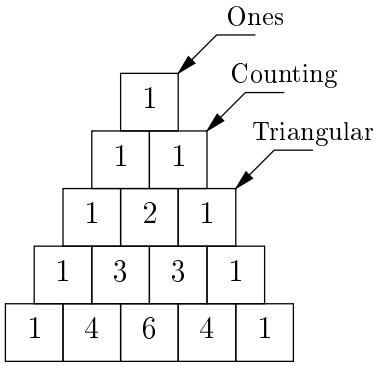
(a) (4 pt) Given a linked list that represents a row in Pascal's triangle, return a linked list that will represent the row below it. See page 2 of the Midterm 2 study guide for the definition of the `Link` class. However, your solution **must not** use `L.__getitem__(k)` (or `L[k]`). You may not need all the lines.

```
from link import *
def pascal_row(s):
    """
    >>> a = Link.empty
    >>> for _ in range(5):
    ...     a = pascal_row(a)
    ...     print(a)
    <1>
    <1 1>
    <1 2 1>
    <1 3 3 1>
    <1 4 6 4 1>
    """
    if s is Link.empty:

        return Link(1)

    start = Link(1)
    last, current = start, s

    while current.rest is not Link.empty:

        last.rest = Link(current.first + current.rest.first)

        last, current = last.rest, current.rest

    last.rest = Link(1)



    return start
```

(b) (4 pt) Fill in the procedure below to create a full Pascal Triangle of height k. Represent the entire triangle as a linked list of the rows of the triangles, which are also linked lists. Again, your solution **must not** use `L.__getitem__(k)` method (or `L[k]`).

```
from link import *
from pascal1_soln import *

def make_pascal_triangle(k):
    """
    >>> make_pascal_triangle(5)
    <<1> <1 1> <1 2 1> <1 3 3 1> <1 4 6 4 1>>
    """

    if k == 0:


        return Link.empty


    row = Link(1)


    end = Link(row)


    result = end


    for _ in range(k-1):


        row = pascal_row(row)


        end.rest = Link(row)


        end = end.rest


    return result
```

(c) (4 pt) Pascal's Triangle contains many patterns within it. For instance, consider the diagonals. The first diagonal (going down the left side) is just a series of 1s. The second diagonal (consisting of the second elements of each row) is the counting numbers. The third diagonal is the triangular numbers.



Fill in the procedure below to take in a Pascal Triangle (represented by a linked list from part b) and return a linked list containing the indicated diagonal. As before, your solution **must not** use `L.__getitem__(k)` (or `L[k]`), and you may not need all the lines.

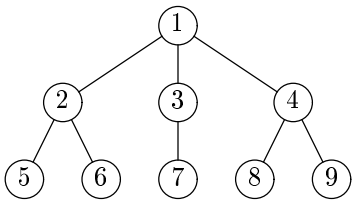
```
from pascal2_soln import *
from link import *
def diagonal(tri, n):
    """
    >>> triangle = make_pascal_triangle(6)
    >>> print(diagonal(triangle, 1))
    <1 1 1 1 1 1>
    >>> print(diagonal(triangle, 2))
    <1 2 3 4 5>
    >>> print(diagonal(triangle, 3))
    <1 3 6 10>
    """
    if tri is Link.empty:


        return Link.empty


    p, j = tri.first, 1


    while j < n and p.rest != Link.empty:


        p, j = p.rest, j + 1


    if p.rest == Link.empty and j != n:


        return diagonal(tri.rest, n)


    return Link(p.first, diagonal(tri.rest, n))
```

(d) (2 pt) Circle the Θ expression that describes the number of integers contained in the value of the expression

`make_pascal_triangle(n)`.

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$ None of these

5. (13 points) Level-Headed Trees A *level-order traversal* of a tree, T, traverses the root of T (level 0), then the roots of all the branches of T (level 1) left to right, then all the roots of the branches of the nodes traversed in level 1, (level 2) and so forth. Thus, a level-order traversal of the tree



visits nodes with labels 1, 2, 3, 4, 5, 6, 7, 8, 9 in that order.

(a) (9 pt) Fill in the following generator function to yield the labels of a given tree in level order. All trees are of the class `Tree`, defined on page 2 of the Midterm 2 Study Guide. The strategy is to use a helper function that yields nodes at one level, and then to call this function with increasing levels until a level does not yield any labels. You may not need all the lines.

```
def level_order(tree):
    """Generate all labels of tree in level order.

    >>> list(level_order(Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])))
    [1, 2, 5, 3, 4]
    """
    def one_level(tree, k):
        """Generate the labels of tree at level k."""

        if  k == 0:

            yield tree.label

        else:

            for child in tree.branches:


                yield from one_level(child, k-1)

    level, count = 0, True

    while count:

        count = 0



        for label in one_level(tree, level):


            count += 1


            yield label


        level += 1
```

(b) (4 pt) Write a function that, given a Python list of values and a tree, returns whether the list contains the labels of the tree in level order. Assume `tree` is an instance of the `Tree` class on your Midterm 2 Study Guide.

```
def same_level_order(tree, s):
    """Return True if and only if list s contains the labels of tree in level order.

    >>> t = Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])
    >>> same_level_order(t, [1, 2, 5, 3, 4])
    True
    >>> same_level_order(t, [1, 2, 3, 4, 5])
    False
    >>> same_level_order(t, [1, 2, 5, 3, 4, 6])
    False
    >>> same_level_order(t, [1, 2, 5, 3])
    False
    """


    k = 0


    for label in level_order(tree):


        if k >= len(s) or s[k] != label:


            return False


        k += 1


    return k == len(s)
```

6. (10 points) Simplify! Simplify! For this problem, consider a very small subset of Scheme containing only **if** expressions, `(if pred then-part else-part)`, and atoms including symbols, `#t` for true, and `#f` for false. Such expressions can be simplified according to the following transformation rules. Here, `P`, `E1`, and `E2` are Scheme expressions in the subset, and `P'`, `E1'`, and `E2'` are their simplified versions.

- The expression `(if P E1 E2)` simplifies to
  - `E1'` if `P'` is `#t`.
  - `E2'` if `P'` is `#f`.
  - `E1'` if `E1'` equals `E2'`.
  - Otherwise, an `if` expression with `P'`, `E1'`, and `E2'` as the predicate, then-part, and else-part.
- Any expression, `E`, simplifies to `#t` if `E` is *known to be true* (see below); or to `#f` if it is *known to be false.*
- Finally, in the expression `(if P E1 E2)`, `P'` is known to be true while simplifying `E1` and is known to be false while simplifying `E2`. Initially, only `#t` is known to be true and only `#f` is known to be false.

Fill in the blanks on the next page so that `(simp E)` returns the simplified version of `E` according to these rules, and the helper function `(simp-context E known-t known-f)` returns the simplification of `E` given that `known-t` is a list of expressions known to be true, and `known-f` is a list of expressions known to be false.

For convenience, assume that `(nth k L)` is defined to return element k of list L (where 0 is the first), and that `(in? E L)` is defined to return true if and only if E is `equal?` to a member of the list L.

```
scm> (simp '(if a b c))
(if a b c)

scm> (simp '(if a b b))
b

scm> (simp '(if #t (if #f a b) c))
b

scm> (simp '(if a (if a b c) (if a d e)))
(if a b e)

scm> (simp '(if (if #t a b) (if a d e) f))
(if a d f)

scm> (simp '(if  (if a b b) (if b c d) (if e f f)))
(if b c f)

scm> (simp '(if (if a b c) (if (if a b c) x y) (if (if a b c) y z)))
(if (if a b c) x z)

scm> (simp '(if (if a b c) (if (if a (if a b b) c) d e) f))
(if (if a b c) d f)
```

```
(define (simp expr)
  (simp-context expr '(#t) '(#f)))
(define (simp-context expr known-t known-f)
  (define simp-expr (if (pair? expr) (simp-if (nth 1 expr) (nth 2 expr) (nth 3 expr) known-t known-f) expr))
  (cond ((in? simp-expr known-t) #t)
        ((in? simp-expr known-f) #f)
        (else simp-expr)))
(define (simp-if pred then-part else-part known-t known-f)
  (let ((simp-pred (simp-context pred known-t known-f)))
    (define simp-then
      (simp-context then-part (cons simp-pred known-t) known-f))
    (define simp-else
      (simp-context else-part known-t (cons simp-pred known-f)))
    (cond ((equal? simp-pred #t) simp-then)
          ((equal? simp-pred #f) simp-else)
          ((equal? simp-then simp-else) simp-then)
          (else (list 'if simp-pred simp-then simp-else)))))
```

7. (10 points) Friendship Consider the table `friends`, defined

```
CREATE TABLE friends AS
  SELECT "Jerry" AS p1, "Neil" AS p2 UNION
  SELECT "Neil"       , "Jerry"      UNION
  SELECT "Neil"       , "John"       UNION
  SELECT "John"       , "Neil"       UNION
  SELECT "John"       , "Paul"       UNION
  SELECT "Paul"       , "John";
```

This particular definition is intended as an example; your code should work for any definition of `friends` in which all pairs of friends appear in both orders and people are not friends of themselves.

(a) (3 pt) Define a table `friends2` containing friends-of-friends (or friends^2). For example, Jerry and Neil are friends, Neil and John are friends, so Jerry and John are friends of friends. Be careful! Jerry is not a second degree friend to himself. The column names should be `p1` and `p2`, as in `friends`.

Expected output:

```
sqlite> SELECT * FROM friends2;
Jerry|John
John|Jerry
Neil|Paul
Paul|Neil
```

```
CREATE TABLE friends2 AS

  SELECT a.p1, b.p2 FROM friends AS a, friends AS b

  WHERE a.p2 = b.p1 AND a.p1 <> b.p2;
```

(b) (7 pt) We could go on to define a table of friends^3 (such as Jerry|Paul and Paul|Jerry), but let's go further and define a table of friends^5 called `friends5` that contains pairs of friends of friends of friends of friends of friends. We want pairs of people who are friends^5 but are not friends, friends^2, friends^3, or friends^4. Our small sample `friends` table has no such pairs, alas, but we can always dream.

To tell that a pair of people are strictly friends^5, we can build a table containing pairs of people plus a "friendship distance" for all distances up to 5. Then we can select just those pairs that appear at distance 5 but never appear at a lesser distance.

```
CREATE TABLE friends5 AS
  WITH distances(p1, p2, dist) AS (
    SELECT p1, p2, 1 from friends UNION
    SELECT d.p1, f.p2, dist+1
      FROM distances AS d, friends AS f
      WHERE d.p2 = f.p1 AND dist < 5
    )
    SELECT p1, p2 FROM distances
      GROUP BY p1, p2 HAVING min(dist) = 5;
```