CS 61A Fall 2016 — 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 two hand-written 8.5" × 11" pages of your own creation and the official CS 61A 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	
Room	
Row & Seat Number	
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)**	

1. (16 points) What Would Python Display?

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. Assume that expressions are evaluated in the order that they appear on the page, with the left column before the right column.

The `Pair` class appears on page 2 of the final study guide. The `reduce` function appears on page 1 of the final study guide. The first two rows have been provided as examples.

Assume that you have started `python3` and executed the following statements:

```
swap = lambda a, b: [b, a]
paws = lambda s: reduce(swap, list(s), 3)
emit = lambda y: [x+y for x in y]
time = lambda t, f: (t and f(t)) or print(t+2)
mite = lambda mu: time(mu + 1, print)
s, t = print, [3 * x for x in range(8)]

def ti(me):
    while me:
        yield time(me, lambda k: [k.pop()])

def newt(graves):
    tina = graves.pop()
    if graves.pop() % 3 == 0:
        graves.pop()
    print(tina, graves.pop())
    if graves:
        return Pair(tina-1, newt(graves))
    return Pair(tina+1, tina)
```


Expression	Interactive Output
`5*5`	25
`print(Pair(6, 1))`	(6 . 1)
`print(2, s(print(3)))`	
`emit([[3], [2]])[1]`	
`s(Pair(Pair(4, 5), nil))`	
`time(2, mite)`	

Expression	Interactive Output
`paws(range(4, 6))`	
`tuple(ti([5, 6]))`	
`s(newt([2, 5, 8]))`	
`s(newt(t))`	

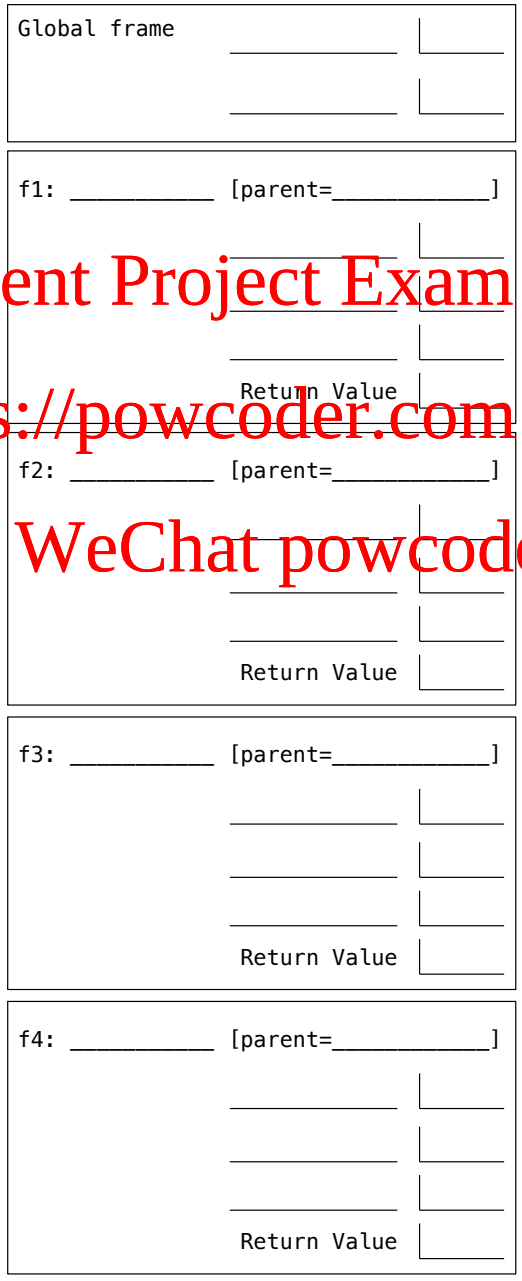
2. (8 points) A Function By Any Other Name

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 lists. You don't need to write index numbers or the word "list".

```
1  name = [2, lambda your: name[0]]
2
3  def your(name):
4      def what(s):
5          nonlocal name
6          name = [name, s]
7          return what is your(name)
8      def your(s):
9          s[1][1](name[:1])
10     return what
11
12 your(True)(name)
```

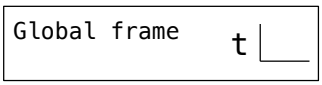


3. (8 points) Boxes and Arrows

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". Please erase or cross out any boxes or pointers that are not part of a final diagram.

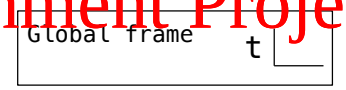
(a)

```
t = [1]
t.append([t, [1]])
```



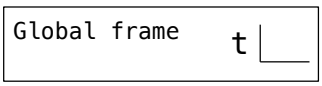
(b)

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



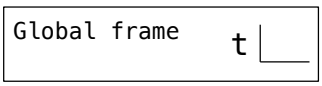
(c)

```
t = [3]
t.extend(t + [t] + [[t]])
```



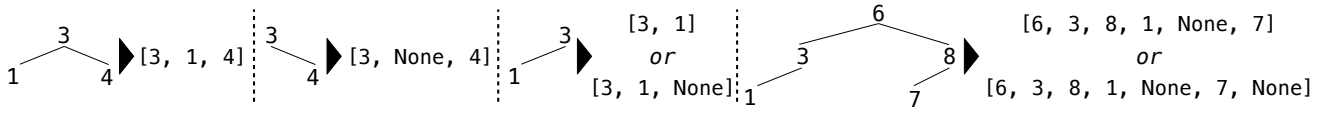
(d)

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



4. (16 points) A Tree or a List

A *flat list* is a list containing integers and `None` that describes a binary tree of integers. The root is element 0. The left branch of element i is element $2*i+1$ and the right is $2*i+2$. The `None` value is used to indicate an empty tree, such as an empty left branch when the corresponding right branch is not empty. `None` values at the end of a flat list can be omitted. Four examples of trees and the flat lists that describe them appear below.



(a) (4 pt) Implement `grow`, which takes a *flat list* `s` of integer node values (which may also contain `None`, as described above) and an index `i` that is `0` by default. When `i` is `0`, `grow` returns a `BTree` instance described by `s`. The `BTree` class appears in the left column of page 2 of the midterm 2 study guide.

```
def grow(s, i=0):
    """Return a binary tree described by the flat list S when I is 0.

    >>> grow([3, 1, 4])
    BTree(3, BTree(1), BTree(4))
    >>> grow([3, None, 4])
    BTree(3, BTree.empty, BTree(4))
    >>> grow([3, 1, None])
    BTree(3, BTree(1))
    >>> grow([6, 3, 8, 1, None, 7])
    BTree(6, BTree(3, BTree(1)), BTree(8, BTree(7)))
    """
    if i >= len(s) or s[i] is None:
        ____________________________________________________________________
    left, right = ____________________, ____________________
    return BTree(_____________________________________________, left, right)
```


(b) (4 pt) Implement `leaves`, which takes a flat list `s` and returns a generator over the leaf values of the tree described by `s`. You may not call `grow` in your solution.

```
def leaves(s):
    """Return a generator over the leaf values in a binary tree described by S.

    >>> list(leaves([3, 1, 4]))
    [1, 4]
    >>> list(leaves([3, 1, None]))
    [1]
    >>> list(leaves([6, 3, 8, 1, None, 7]))
    [1, 7]
    """
    f = ____________________________________________________________________
    for i in ____________________________________________________________:
        if ___________________________________ and f(2*i+1) and f(2*i+2):
            ____________________________________________________________
```

(c) (2 pt) Circle the Θ expression that describes the height of the tree returned by `grow(s)` for an `s` of length n that ends with an integer. The height of a tree is the length of the longest path from its root to a leaf.

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

(d) (6 pt) A `Set` instance represents a set of integers as a binary search tree. The node values are stored in a flat list called `items`. Implement the `find` method, which returns whether an element is in the `Set` and if not, adds it as a new leaf if the argument `add` is `True`. *Hint*: `int(True)` is 1 and `int(False)` is 0.

```
class Set:
    """A set of ints stored in a binary search tree described by a flat list.

    >>> s = Set()
    >>> [s.add(k) for k in [6, 3, 1, 8, 9, 4, 4, 4]]
    [False, False, False, False, False, False, True, True]
    >>> s.items                # The node values of a binary search tree
    [6, 3, 8, 1, 4, None, 9]
    >>> [s.has(k) for k in range(10)]
    [False, True, False, True, True, False, True, False, True, True]
    >>> [s.add(3), s.add(2)]  # 3 was already in the set; 2 was added
    [True, False]
    >>> s.items                # 2 appears as a leaf; the right branch of 1.
    [6, 3, 8, 1, 4, None, 9, None, 2]
    """
    def __init__(self):
        self.items = []
    def add(self, v):
        """Ensure that v is in this set; return whether it was already there."""
        return self.find(v, True)
    def has(self, v):
        """Return whether v is in the set."""
        return self.find(v, False)

    def find(self, v, add):

        i = 0

        while ____________________________________________________________:

            if ____________________________________________________________:

                return True

            i = 2 * i + ____________________________________________________

        if add:

            self.items += ______________________________________________

            self.items[i] = ____________________________________________

        return False
```

5. (10 points) Reset

(a) (6 pt) Implement the `fset` function, which returns two functions that together represent a set. Both the `add` and `has` functions return whether a value is already in the set. The `add` function also adds its argument value to the set. You may assign to only one name in the assignment statement. You may not use any built-in container, such as a set or dictionary or list.

```
def fset():
    """Return two functions that together represent a set.

    >>> add, has = fset()
    >>> [add(1), add(3)]              # Neither 1 nor 3 were already in the set
    [False, False]
    >>> [has(k) for k in range(5)]
    [False, True, False, True, False]
    >>> [add(3), add(2)]              # 3 was already in the set; 2 is added
    [True, False]
    >>> [has(k) for k in range(5)]
    [False, True, True, True, False]
    """

    items = lambda x: ____________________________________________________
    def add(y):

        ____________________________________________________________________

        f = items

        _________ = ________________________________________________________

        return ____________________________________________________________


    return add , ___________________________________________________________
```


(b) (4 pt) Implement the `cycle` procedure, which takes a non-empty Scheme list of values. It returns an infinite stream that repeats those values in order, indefinitely. For example, `(cycle '(6 1 a))` evaluates to the infinite stream containing the values `6 1 a 6 1 a 6 1 a 6 1 ...`

```
(define (cycle s)

    (define (with t)

        (if (null? t)


            ____________________________________________________________


            (cons-stream ________________________  ________________________))))


    ____________________________________________________________________)
```

6. (10 points) I Scheme for Ice Cream

The built-in `append` procedure is equivalent in behavior to the following definition.

```
(define (append s t) (if (null? s) t (cons (car s) (append (cdr s) t))))
```

(a) (1 pt) Circle *True* or *False*: The recursive call to `append` in the definition above is a tail call.

(b) (4 pt) Implement `atoms`, which takes a Scheme expression. It returns a list of the non-`nil` atoms contained in the expression in the order that they appear. A non-`nil` atom is a number, symbol, or boolean value.

```
scm> (atoms 1)
(1)
scm> (atoms '(+ 2 3))
(+ 2 3)
scm> (atoms '(+ (* 2 3) 4))
(+ * 2 3 4)
scm> (atoms '(* (+ 1 (* 2 3)) (+ 4 5)))
(* + 1 * 2 3 + 4 5)
```

```
(define (atoms exp)

    (cond ((null? exp) _____________________________________________________ )

          ((atom? exp) _____________________________________________________ )

          (else (append  ________________________  ________________________ ))))
```


(c) (5 pt) If Scheme had only numbers and two-argument procedures, parentheses would be unnecessary. To demonstrate, implement `tally`, which takes the list of atoms in a Scheme expression. It returns a list whose first element is the value of the original expression. Assume that the original expression consists only of numbers and call expressions with arithmetic operators (such as `+` and `*`) and exactly two operands. *Hint*: `tally` is similar to the built-in `eval` procedure: `(eval '(+ (* 2 3) 4))` evaluates to 10.

```
scm> (car (tally '(1)))                  ; atoms in 1
1
scm> (car (tally '(+ 2 3)))              ; atoms in (+ 2 3)
5
scm> (car (tally '(+ * 2 3 4)))          ; atoms in (+ (* 2 3) 4)
10
scm> (car (tally '(* + 1 * 2 3 + 4 5))) ; atoms in (* (+ 1 (* 2 3)) (+ 4 5))
63
```

```
(define (tally s)

  (if (number? (car s))  ____________________________________________________

      (let ((first  ____________________________________________________ ))

        (let ((second  ____________________________________________________ ))

          (cons ( ____________ ( ____________ (car s) (car first) (car second)))

                ____________________________________________________ )))))
```

7. (12 points) Hailstoned

The following two tables of integers from 0 to 99 (including 99) and 1 to 99 are used in the questions below.

```
create table ns as
  with t(n) as ( select 0 union select n+1 from t where n < 99 )
  select * from t;
create table ps as select n from ns where n > 0;
```

(a) (3 pt) Create a one-column table of all integers from 2000 to 9999 (including 9999). Do not use recursion.

```
select ____________________ from ____________________ where ____________________ ;
```

(b) (5 pt) Create a table with a row for each pair of different positive integers x and y below 100. The three columns contain x, y, and the greatest common divisor of x and y. For example, one row contains (20, 50, 10) because 10 is the largest n that evenly divides both 20 and 50. Another row contains (50, 20, 10).

```
select _____________ as x, _____________ as y, _____________ as gcd

  from ____________________________________________________________
  where ____________________________________________________________
  group by _____________________________________________________ ;
```


(c) (4 pt) A *hailstone sequence* begins with a positive m. If m is even, divide it by 2. If m is odd, triple it and add 1. Repeat until 1 is reached. For example, the sequence starting at 3 is 3, 10, 5, 16, 8, 4, 2, 1. Create a two-column table with positive integer m in the first column and the length of the hailstone sequence starting at m in the second column. For example, one row contains (3, 8) because the hailstone sequence starting at 3 has 8 elements. This table should have one row for each m, but only include m for which the entire hailstone sequence starting at m consists of numbers below 100.


```
with hailstone(m, length) as (

  select 1, 1 union

  select ___________, ___________ from ____________________________________

    where (_____________________________________________________________) or

          (_____________________________________________________________)

) select * from hailstone;
```

8. (0 points) Draw! *(Optional)* Compare Python, Scheme, and SQL in a picture.