CS 61A Fall 2015

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 one hand-written 8.5" × 11" crib sheet 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	
BearFacts email (`_@berkeley.edu`)	
TA	
Room	
Seat	
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. (10 points) From the Other Side

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 exception is raised, write "Exception". If evaulation would run forever, write "Forever". If an iterator or generator value would be displayed, write "Iterator" (instead of something like `<str_iterator object at 0x...>`).

The first three rows have been provided as examples.

Assume that you have started `python3` and executed the following statements (which do not cause errors):

```
class Adele:
    times = '1000'
    def __init__(self, you):
        self.call = you
    def __str__(self):
        return self.times

class Hello(Adele):
    def __next__(self):
        return next(self.call)

never = iter('scheme2Bhome')

def any(more):
    next(never)
    print(outside)
    yield next(never)
    print(next(never))
    yield more(more)

outside = Hello(any(any))
```

Expression	Interactive Output
`'a'`	`'a'`
`iter('a')`	Iterator
`print('a') + 1`	`a` Exception
`next(never)`	`'s'`
`next(outside)`	1000 `'h'`
`next(next(outside))`	e 1000 `'e'`
`list(never)[:3]`	`['2', 'B', 'h']`
`next(next(outside))`	Exception

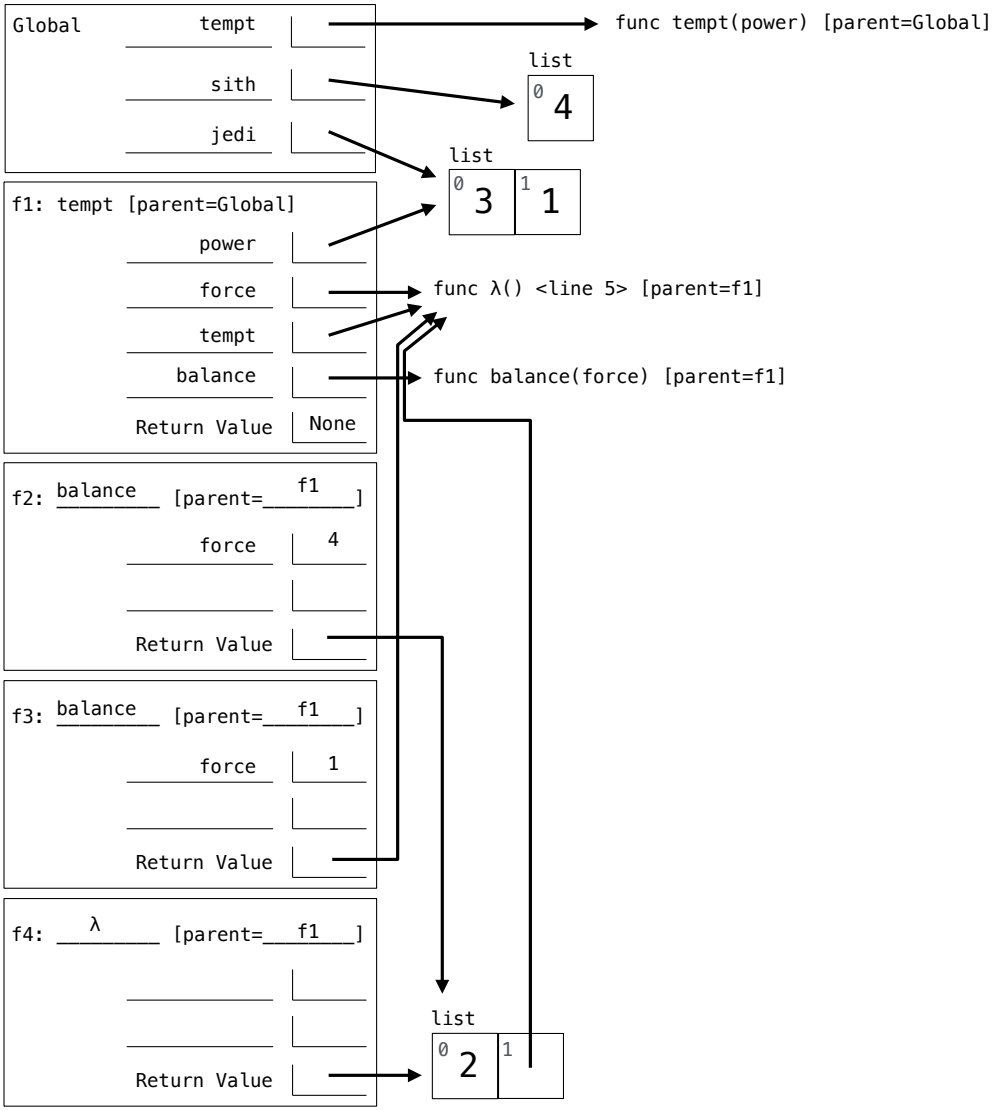
2. (16 points) Endor

(a) (8 pt) 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:

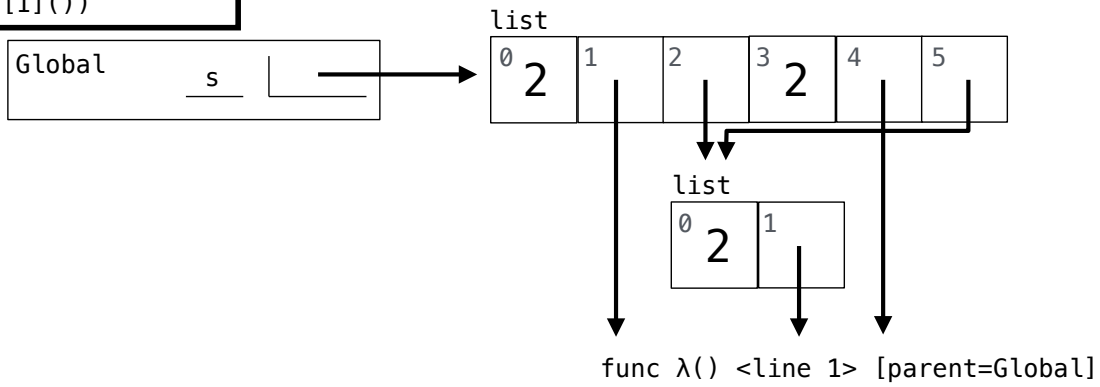
- Add all missing names and parent annotations to all frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
1  sith = []
2  jedi = [3]
3
4  def tempt(power):
5      force = lambda: [len(power), force]
6      tempt = force
7      def balance(force):
8          nonlocal power
9          power.append(force)
10         if force < 2:
11             return tempt
12         power = jedi
13         return balance(1)()
14     balance(4)
15
16 tempt(sith)
```

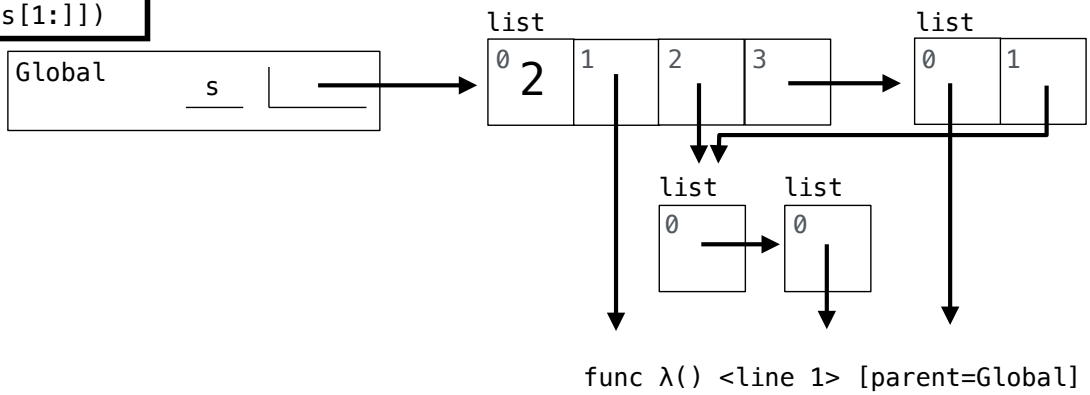


(b) **(8 pt)** Fill in the value for s in the global frame that results from executing each block of code below. *You do* ***not*** *need to draw any local lambda frames.* You must use box-and-pointer diagrams for full credit. The `list` methods `append`, `extend`, and `pop` are demonstrated on the midterm 2 study guide, page 2, right column. *Hint:* You may want to write your answer on scratch paper first in order to avoid messy diagrams.

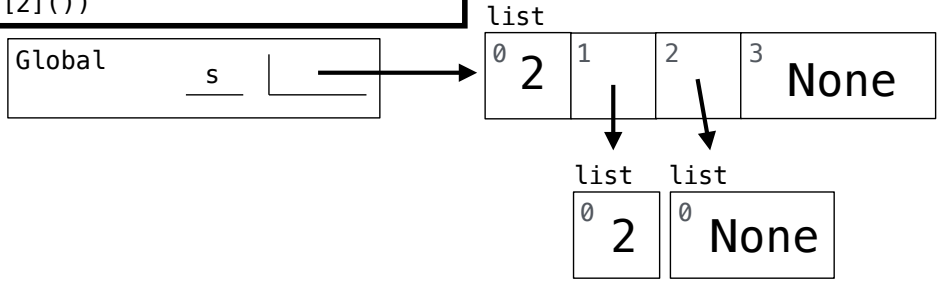
```
1 s = [2, lambda: list(s)]
2 s.append(s[1]())
3 s.extend(s[1]())
```



```
1 s = [2, lambda: s]
2 s.append([s[1:]])
3 s.extend([s[1:]])
```



```
1 s = [2, lambda: list(s)]
2 s.append(lambda: [s.append(s.pop()())])
3 s.extend(s[2]())
```



3. (10 points) Forest Path

Definition. A *path* through a `Tree` is a list of adjacent node values that starts with the root value and ends with a leaf value. For example, the paths of `Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])` are

```
[1, 2]
[1, 3, 4]
[1, 3, 5]
```

The `Tree` class is defined on the midterm 2 study guide. The `one` function defined below is used in the questions below to convert true and false values into the numbers 1 and 0, respectively.

```
def one(b):
    if b:
        return 1
    else:
        return 0
```

(a) (3 pt) Implement `bigpath`, which takes a `Tree` instance `t` and an integer `n`. It returns the number of paths in `t` whose sum is *at least* `n`. Assume that all node values of `t` are integers.

```
def bigpath(t, n):
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> bigpath(t, 3)
    3
    >>> bigpath(t, 6)
    2
    >>> bigpath(t, 9)
    1
    """
    if t.is_leaf():


        return one(t.entry >= n)


    return sum([bigpath(b, n-t.entry) for b in t.branches])
```

(b) (2 pt) Circle the Θ expression that describes the number of paths in a tree with n nodes in which every non-leaf node has at least 2 branches.

$\Theta(1)$ $\Theta(\log n)$ $\Theta(n)$ $\Theta(n^2)$ $\Theta(2^n)$

Definition. A *path* through a `Tree` is a list of adjacent node values that starts with the root value and ends with a leaf value. For example, the paths of `Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])` are

```
[1, 2]
[1, 3, 4]
[1, 3, 5]
```

(c) (3 pt) Implement `allpath` which takes a `Tree` instance `t`, a one-argument predicate `f`, a two-argument reducing function `g`, and a starting value `s`. It returns the number of paths `p` in `t` for which `f(reduce(g, p, s))` returns a true value. The `reduce` function is on the final study guide. You do not need to call it, though.

```
def allpath(t, f, g, s):
    """Return the number of paths p in t for which f(reduce(g, p, s)) is true.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> even = lambda x: x % 2 == 0
    >>> allpath(t, even, max, 0)  # Path maxes are 2, 4, and 5
    2
    >>> allpath(t, even, pow, 2)  # E.g., pow(pow(2, 1), 2) is even
    3
    >>> allpath(t, even, pow, 1)  # Raising 1 to any power is odd
    0
    """
    if t.is_leaf():


        return one(f(g(s, t.entry)))


    return sum([allpath(b, f, g, g(s, t.entry)) for b in t.branches])
```

(d) (2 pt) Re-implement `bigpath` (part a) using `allpath` (part c). Assume `allpath` is defined correctly.

```
from operator import add, mul

def bigpath(t, n):
    """Return the number of paths in t that have a sum larger or equal to n.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> bigpath(t, 3)
    3
    >>> bigpath(t, 6)
    2
    >>> bigpath(t, 9)
    1
    """
    return allpath(t, lambda x: x>=n, add, 0)
```

4. (8 points) Cucumber

Cucumber is a card game. Cards are positive integers (no suits). Players are numbered from 0 up to `players` (0, 1, 2, 3 in a 4-player game). In each `Round`, the players each `play` one card, starting with the `starter` and in ascending order (player 0 follows player 3 in a 4-player game). If the `card` played is as high or higher than the `highest` card played so far, that player takes `control`. The winner is the last player who took control after every player has played once. Implement `Round` so that `play_round` behaves as described in the doctests below. Part of your score on this question will be assigned based on *composition* (don't repeat yourself).

```
def play_round(starter, cards):
    """Play a round and return all winners so far. Cards is a list of pairs.
    Each (who, card) pair in cards indicates who plays and what card they play.

    >>> play_round(3, [(3, 4), (0, 8), (1, 8), (2, 5)])
    [1]
    >>> play_round(1, [(3, 5), (1, 4), (2, 5), (0, 8), (3, 7), (0, 6), (1, 7)])
    It's not your turn, player 3
    It's not your turn, player 0
    The round is over, player 1
    [1, 3]
    >>> play_round(3, [(3, 7), (2, 5), (0, 9)]) # Round is never completed
    It's not your turn, player 2
    [1, 3]
    """
    r = Round(starter)
    for who, card in cards:
        try:
            r.play(who, card)
        except AssertionError as e:
            print(e)
    return Round.winners

class Round:
    players, winners = 4, []
    def __init__(self, starter):
        self.starter, self.player, self.highest = starter, starter, -1

    def play(self, who, card):

        assert not self.complete(), 'The round is over, player '+str(who)

        assert who == self.player, "It's not your turn, player "+str(who)

        self.player = (who + 1) % self.players

        if card >= self.highest:

            self.highest, self.control = card, who

        if self.complete():

            self.winners.append(self.control)

    def complete(self):

        return self.player == self.starter and self.highest > -1
```

5. (14 points) Grouper

(a) (4 pt) Implement `group`, which takes a one-argument function `f` and a list `s`. It returns a list of groups. Each group is a list that contains all the elements `x` in `s` that return equal values for `f(x)`. The elements in a group appear in the same order that they appeared in `s`. The groups are ordered by the order in which their first elements appeared in `s`.

```
def group(f, s):
    """Return a list of groups that contain all x with equal f(x).

    >>> five = [3, 4, 5, 2, 1]
    >>> group(lambda x: x % 2, five)
    [[3, 5, 1], [4, 2]]
    >>> group(lambda x: x % 3, five)
    [[3], [4, 1], [5, 2]]
    """
    a = []
    for b in map(f, s):


        if b not in a:


            a.append(b)


    return [[x for x in s if f(x)==b] for b in a]
```

(b) (4 pt) Implement `group_link`, which takes a one-argument function `f` and a `Link` instance `s`. It returns a linked list of groups. Each group is a `Link` instance containing all the elements `x` in `s` that return equal values for `f(x)`. The order of groups and elements is the same as for `group`. The `Link` class appears on your midterm 2 study guide. The `filter_link` function appears in the appendix on the last page of this exam.

```
def group_link(f, s):
    """Return a linked list of groups that contain all x with equal f(x).

    >>> five = Link(3, Link(4, Link(5, Link(2, Link(1)))))
    >>> group_link(lambda x: x % 2, five)
    Link(Link(3, Link(5, Link(1))), Link(Link(4, Link(2))))
    >>> group_link(lambda x: x % 3, five)
    Link(Link(3), Link(Link(4, Link(1)), Link(Link(5, Link(2)))))
    """
    if s is Link.empty:
        return s
    else:
        a = filter_link(lambda x: f(x)==f(s.first), s)


        b = filter_link(lambda x: f(x)!=f(s.first), s) # s.rest ok too


        return Link(a, group_link(f, b))
```

Definition. The *multi-grouping* by function `f` of list `s` is formed by the following iterative process with `k` starting at 0 and increasing by 1 each iteration.

- Group together all elements that yield equal values when applying `f` repeatedly, `k` times.
- If all elements are in a single group, the process is complete. Otherwise, place each new group in a (possibly nested) list and repeat.

For example, if `f` is `lambda x: max(x-3, 0)` and `s` is `[2, 4, 3, 4, 2]`, then

- In the `k=0` iteration, the 2's are grouped, the 4's are grouped, and the 3 is alone: `[[2, 2], [3], [4, 4]]`
- In the `k=1` iteration, `f(2)==f(3)`, and so the 2's group and 3's group are grouped: `[[[2, 2], [3]], [[4, 4]]]`
- In the `k=2` iteration, `f(f(2))==f(f(3))==f(f(4))`. All elements are in a single group, so we're done.

(c) (4 pt) Implement `multigroup`, which returns the multi-grouping by `f` of a list `s`. Assume that the process terminates and that `group` (part a) is implemented correctly.

```
def multigroup(f, s):
    """Return a multi-grouping by f of the elements in s.

    >>> multigroup(lambda x: max(x-3, 0), [2, 4, 3, 4, 2])
    [[[2, 2], [3]], [[4, 4]]]
    >>> multigroup(abs, [5])
    5
    >>> multigroup(abs, [5, 5])
    [5, 5]
    >>> multigroup(abs, [5, 5, -5])
    [[5, 5], [-5]]
    >>> multigroup(lambda x: x // 10, [123, 145, 126, 149])
    [[[123], [126]], [[145], [149]]]
    >>> multigroup(lambda x: x[1:], ['tin', 'man', 'can'])
    [[['tin']], [['man'], ['can']]]
    >>> multigroup(lambda x: max(x-1, 0), [2, 4, 3, 4, 2])
    [[[[2, 2]]], [[[3]]], [[[[4, 4]]]]]
    """
    def using(g, s):

        if len(s) == 1:

            return s[0]

        else:

            grouped = group(g, s)

            return using(lambda x: f(g(x[0])), grouped)

    return using(lambda x: x, s)
```

(d) (2 pt) How many square brackets are in the return value of `multigroup(hail, [3, 20, 128])`? Assume that `multigroup` is implemented correctly.

```
def hail(x):
    if x == 1:
        return 1
    elif x % 2 == 0:
        return x // 2
    else:
        return 3 * x + 1
```

16: `[[[[3], [20]]], [[[128]]]]`

6. (10 points) Pair Emphasis

(a) (6 pt) Implement `parens` by crossing out whole lines below. It takes a Scheme value and returns the number of parentheses required to write the value in standard Scheme notation.

```
; Return the number of parentheses in s.
;
; (parens 3)           -> 0
; (parens (list 3 3))  -> 2
; (parens '(3 . 3))    -> 2
; (parens '(3 . (3)))  -> 2  because (3 . (3)) simplifies to (3 3)
; (parens '((3)))      -> 4
; (parens '(()))       -> 4
; (parens '((3)((3)))) -> 8
(define (parens s) (f s 2))
(define (f s t)
    (cond ((pair? s) (+
            t
            (f (car s) 2)
            (f (cdr s) 0)))
          ((null? s) t)
          (else 0)))
```

(b) (2 pt) Write a quote expression that evaluates to the Scheme list `(1 (2) 3)` that has as many parentheses as possible in the expression. For example, a well-formed (but incorrect) answer is `(quote (1 (2) 3))`.

```
(quote (1 . ((2 . ()) . (3 . ()))))
```

(c) (2 pt) How many total calls to `scheme_eval` would be required to evaluate `(parens 3)` in your Scheme interpeter (Project 4)? Assume that `parens` is implemented correctly. Assume you are **not** using the tail-recursive `scheme_optimized_eval`.

15

7. (12 points) Highly Exclusive

(a) (4 pt) Complete the definition of `no-fib`, the stream of all positive integers that are not Fibonacci numbers. These are all positive integers excluding 1, 2, 3, 5, 8, 13, ... The stream starts with 4, 6, 7, 9, 10, 11, 12, 14.

```
(define (p prev curr n)

    (if (= n curr)

        (p curr (+ prev curr) (+ n 1))

        (cons-stream n (p prev curr (+ n 1)))))

(define no-fib (p 3 5 4))
```

(b) (4 pt) A Hamming number is a positive integer that has no prime factors other than 2, 3, or 5. That is, all Hamming numbers are `pow(2, i) * pow(3, j) * pow(5, k)` for some non-negative integers `i`, `j`, and `k`. The first 20 Hamming numbers are 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, and 36.

Complete the SQL statements below so that the final statement generates a single-column table that contains as its rows the Hamming numbers less than 100 in increasing order.

```
create table t as select 2 as k union select 3 union select 5;

with ham(n) as (

    select 1 union

    select n * k from ham, t where n * k < 100

) select n from ham order by n;
```

(c) (4 pt) Select all positive integers that have at least 3 proper multiples that are less than or equal to X. A proper multiple m of n is an integer larger than n such that n evenly divides m (`m % n == 0`).

The resulting table should have two columns. Each row contains an integer (that has at least 3 proper multiples) and the number of its proper multiples up to X. For example, the integer 3 has 5 proper multiples up to 20: 6, 9, 12, 15, and 18. Therefore, `3|5` is a row. There are five rows in the table when X is 20: `1|19`, `2|9`, `3|5`, `4|4`, and `5|3`. Your statement must work correctly even if X changes to another constant (such as 30) to receive full credit.

```
create table X as select 20 as X;
with ints(n) as (select 1 union select n+1 from ints, X where n < X)

select b.n, count(*) from ints as a, ints as b

       where a.n > b.n and a.n % b.n = 0

       group by b.n having count(*) > 2;
```

8. (0 points) Draw! *(Optional)* In 2050, what will a CS 61A teaching assistant (GSI/UGSI) look like?

Appendix. *This is not a question.*

```
def filter_link(f, s):
    """Return a Link with the elements of s for which f returns a true value."""
    if s is Link.empty:
        return s
    else:
        filtered = filter_link(f, s.rest)
        if f(s.first):
            return Link(s.first, filtered)
        else:
            return filtered
```