CS 61A Spring 2015

Structure and Interpretation of Computer Programs

FINAL EXAM 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 3 official 61A midterm study guides attached to the back of this exam.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation.

Last name	
First name	
SID	
Login	
TA & section time	
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)**	

For staff use only

Q. 1	Q. 2	Q. 3	Q. 4	Q. 5	Q. 6	Total
/16	/14	/8	/18	/14	/10	/80

THIS PAGE CONTAINS NO QUESTIONS

For your reference, a complete implementation of the `Tree` class appears below.

```
class Tree:
    """A tree with entry as its root value.

    >>> Tree(1)
    Tree(1)
    >>> Tree(1, [])
    Tree(1)
    >>> Tree(Tree(1))
    Tree(Tree(1))
    >>> t = Tree(1, [Tree(2), Tree(3)])
    >>> t
    Tree(1, [Tree(2), Tree(3)])
    >>> t.entry
    1
    >>> t.branches
    [Tree(2), Tree(3)]
    """
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def __repr__(self):
        if self.branches:
            branches_str = ', ' + repr(self.branches)
        else:
            branches_str = ''
        return 'Tree({0}{1})'.format(self.entry, branches_str)
```

1. (16 points) Lumberjack

For each row below, fill in the blanks in the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and **expressions may affect later expressions**.

Whenever the interpreter would report an error, write ERROR. You *should* include any lines displayed before an error. *Reminder*: The interactive interpreter displays the `repr` string of the value of a successfully evaluated expression, unless it is `None`. The first two rows are completed for you.

The `Tree` class appears on the previous page. It's also on the Midterm 2 Study Guide, but the exam version is complete. Assume you have started Python 3, executed the `Tree` class statement, then executed the following:

```
odd = lambda x: (x % 2) == 1
big = lambda x: x > 2
f = lambda f: lambda g: lambda x: not (f(x) or g(x))

def choose(s, f):
    for i in range(0, len(s)):
        if f(s[i]):
            return Tree(i, [Tree(j) for j in s[i:] if not f(j)])

lumber = Tree(1)
jack = Tree(1, [lumber, lumber])
lumber.entry = 2
lumber = Tree(lumber)

class Wood(Tree):
    def __init__(self, source):
        source.entry += 1
        Tree.__init__(self, Tree(source))
```

Expression	Output
`Tree(2, [Tree(3)])`	`Tree(2, [Tree(3)])`
`1/0`	ERROR
`tuple(map(odd, filter(big, range(5))))`	`(True, False)`
`sum(filter(f(big)(print), range(1, 5)))`	1 2 3
`lumber`	`Tree(Tree(2))`
`jack.branches`	`[Tree(2), Tree(2)]`
`print(choose([1], big))`	`None`
`choose([2, 3, 4], odd)`	`Tree(1, [Tree(4)])`
`choose([4, 3, 2], f(big)(odd))`	`Tree(2)`
`Wood(Tree(2))`	`Tree(Tree(Tree(3)))`

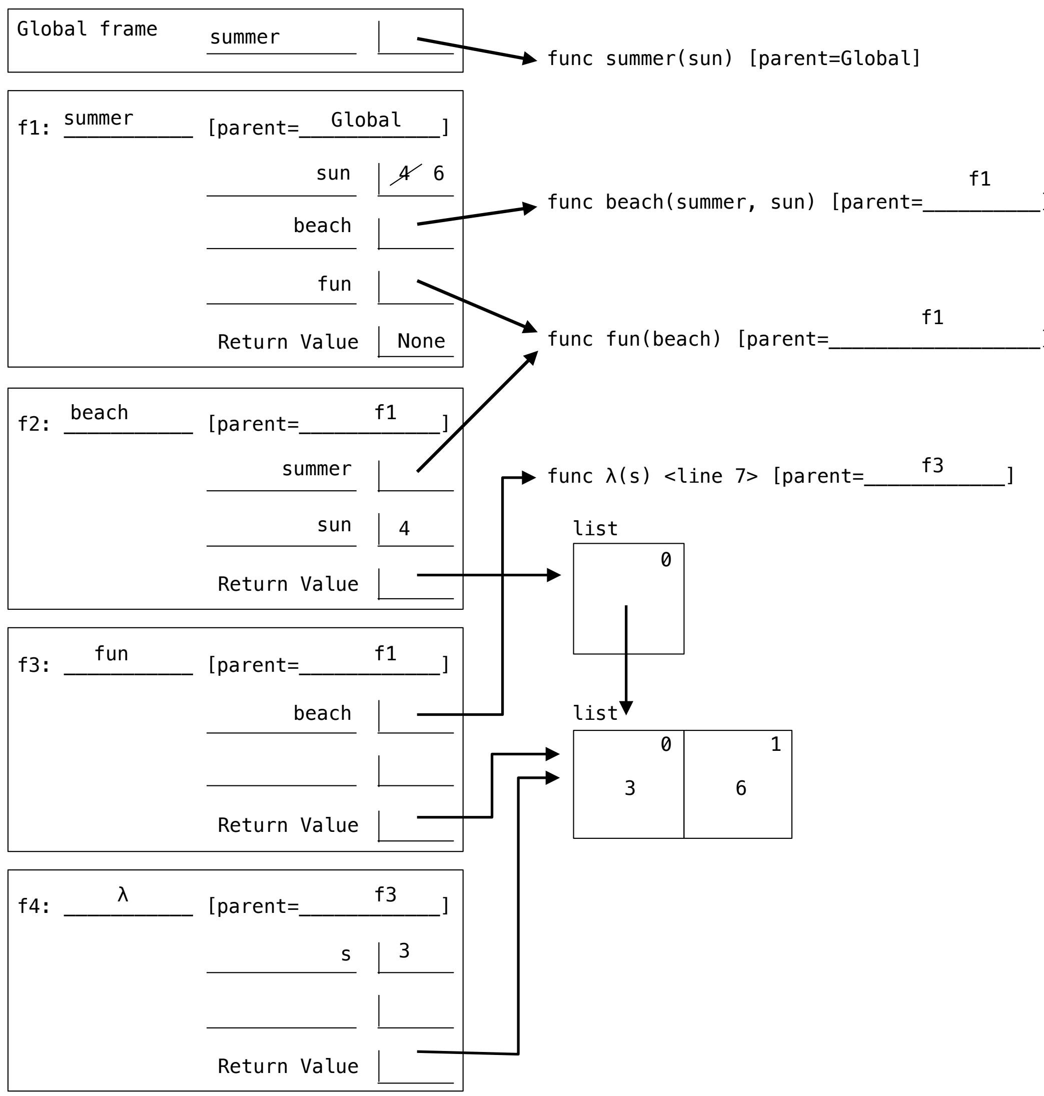
2. (14 points) Hot and Cold

(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 local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.

```
1  def summer(sun):
2      def beach(summer, sun):
3          return [summer(sun+1)]
4      def fun(beach):
5          nonlocal sun
6          sun = sun + 2
7          beach = lambda s: [s, sun]
8          return beach(3)
9      beach(fun, sun)
10 summer(4)
```

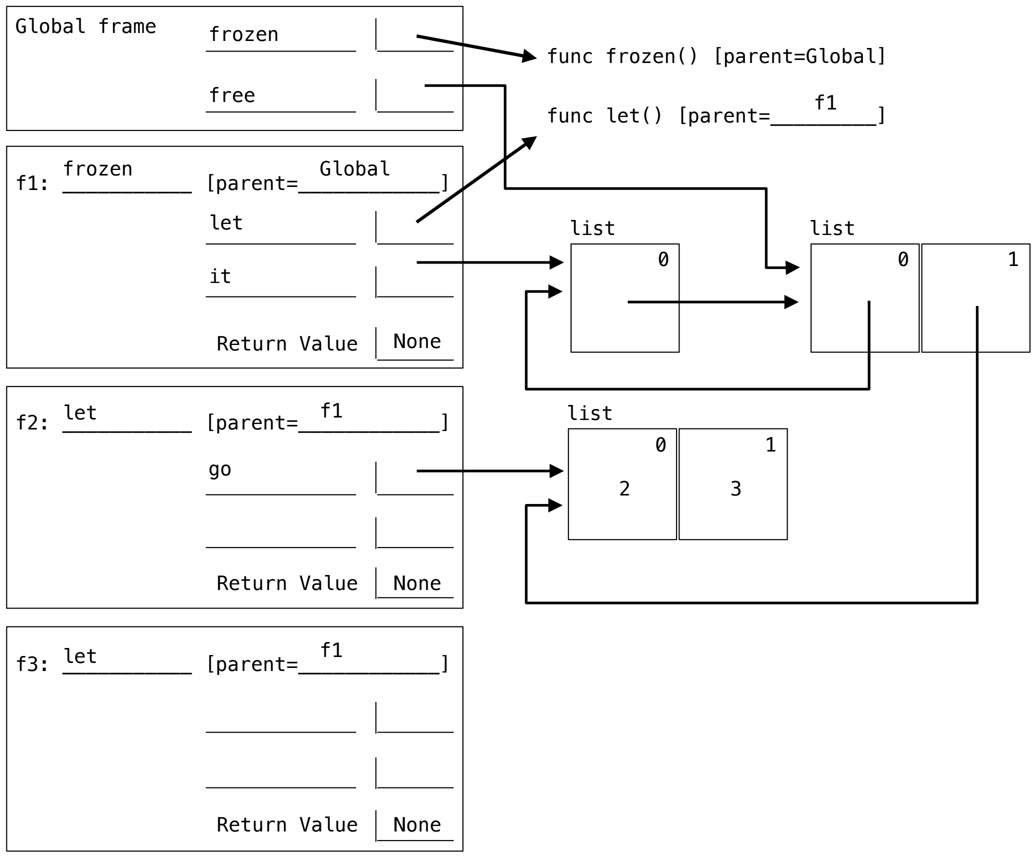


(b) **(6 pt)** Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* Examples of using the `append` and `extend` methods of lists appear on the Midterm 2 Study Guide.

A complete answer will:

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

```
def frozen():
    def let():
        nonlocal it
        if it:
            go, it = it[0], it[1:]
            free.extend([it, go])
            let()
    it = [[2, 3]]
    let()
    it.append(free)
free = []
frozen()
```



3. (8 points) Team Stream

When you complete all the questions on this page, all of the doctests of `append` should pass.

```
def append(s, t):
    """Return a stream with the elements of s followed by the elements of t.

    >>> bits = Stream(0, lambda: Stream(1))    # 0, 1
    >>> taipei = append(bits.rest, bits)       # 1, 0, 1
    >>> taipei.first, taipei.rest.first, taipei.rest.rest.first
    (1, 0, 1)
    >>> repeat(bits).rest.rest.rest.first      # 0, 1, 0, (1), 0, ...
    1
    >>> repeat(bits).rest.rest.rest.rest.first # 0, 1, 0, 1, (0), ...
    0
    >>> ten = unique(taipei) # 1, 0
    >>> ten.first, ten.rest.first, ten.rest.rest
    (1, 0, Stream.empty)
    """
    if s is Stream.empty:
        return t
    return Stream(s.first, lambda: append(s.rest, t))
```

(a) (2 pt) Implement `repeat`, which takes a non-empty stream `s` that is an instance of the `Stream` class. It returns an infinite stream that cycles through the elements of `s` in order. You may use the `append` function.

```
def repeat(s):
    """Return a stream that cycles infinitely through the elements of s."""

    return Stream(s.first,

                  lambda: append(s.rest, repeat(s)))
```

(b) (2 pt) Implement `unique`, which takes a finite stream `s` that is either `Stream.empty` or a `Stream` instance. It returns a stream of the unique elements in `s`. `filter_stream` appears on the Final Study Guide.

```
def unique(s):
    """Return a stream of the unique elements of s in the order that they
    first appear in s. The result contains no repeated elements."""

    if s is Stream.empty:

        return s

    t = filter_stream(lambda x: x != s.first, s.rest)

    return Stream(s.first, lambda: unique(t))
```

(c) (2 pt) Circle **all** Θ expressions below that describe the total number of elements in the stream `append(s, t)` for a finite stream `s` of length `m` and a finite stream `t` of length `n`.

$\Theta(1)$ $\Theta(m)$ $\Theta(n)$ $\Theta(m+n)$ $\Theta(m*n)$

(d) (2 pt) Circle **all** Θ expressions below that describe the total number of function calls required to evaluate `append(s, t)` for a finite stream `s` of length `m` and a finite stream `t` of length `n`.

$\Theta(1)$ $\Theta(m)$ $\Theta(n)$ $\Theta(m+n)$ $\Theta(m*n)$

4. (18 points) Apply That Again

(a) (4 pt) Implement `amplify`, a generator function that takes a one-argument function `f` and a starting value `x`. The element at index k that it yields (starting at 0) is the result of applying `f` k times to x. It terminates whenever the next value it would yield is a false value, such as `0`, `''`, `[]`, `False` etc.

```
def amplify(f, x):
    """Yield the longest sequence x, f(x), f(f(x)), ... that are all true values.

    >>> list(amplify(lambda s: s[1:], 'boxes'))
    ['boxes', 'oxes', 'xes', 'es', 's']
    >>> list(amplify(lambda x: x//2-1, 14))
    [14, 6, 2]
    """

    while x:

        yield x

        x = f(x)
```

(b) (6 pt) Answer the following three questions about the `echo` function below, which you should try to understand by reading its implementation. *Assume that `amplify` is implemented correctly.*

```
def echo():
    x = 0
    def gecko(y):
        nonlocal x
        x = x + 1
        return y - x
    return gecko
```

Circle the value of `echo()(echo()(5))`.

1   2   3   4   5

Circle *all values* of `n` below for which the expression `list(amplify(echo(), n))` terminates.

5   10   15   20   25

Write the largest integer `n` less than 40 for which the expression `list(amplify(echo(), n))` terminates

36

*****Definition***:** A *shrinking function* `f` is a function for which `f(x) < x` for all positive integers `x`. In addition, for any `x`, applying `f` a finite number of times will result in a number less than or equal to 0.

(c) (4 pt) Implement `near_zero`, which takes a shrinking function `f` and a positive starting number `x`. It returns the **smallest positive value** that results from applying `f` to `x` zero or more times. *Assume that `amplify` (previous page) is implemented correctly. You may use it in your solution.*

```
def near_zero(f, x):
    """Return the value nearest zero obtained by repeatedly applying the
    shrinking function f to a non-zero number x.

    >>> [near_zero(lambda x: x-4, k) for k in [3, 4, 5, 6, 7, 8, 9]]
    [3, 4, 1, 2, 3, 4, 1]
    """

    last = x


    for v in amplify(f, x):


        if v < 0:

            return last

        last = v

    return last
```

(d) (4 pt) Implement `count_sums` which counts the number of ways that a positive integer `n` can be partitioned into a subset of the positive values `m, f(m), f(f(m)), ...` for a shrinking function `f`. **No negative values or repeated values can be included in the sum.**

```
def count_sums(n, f, m):
    """Return the number of ways that n can be partitioned into unique positive
    values obtained by applying the shrinking function f repeatedly to m.

    >>> count_sums(6, lambda k: k-1, 4) # 4+2, 3+2+1
    2
    >>> count_sums(12, lambda k: k-2, 12) # 12, 10+2, 8+4, 6+4+2
    4
    >>> count_sums(11, lambda k: k//2, 8) # 8+2+1
    1
    """
    if n == 0:
        return 1
    elif m <= 0 or n < 0:
        return 0
    else:

        yes = count_sums(n-m, f, f(m))


        no = count_sums(n, f, f(m))

        return yes + no
```

5. (14 points) Treasure Hunt

(a) (4 pt) *Leap*rechauns leap from one pot-o'-gold to the next, collecting treasure. However, they can't collect two adjacent pots. Implement `leap`, which takes a list of `pots`. It returns the maximal sum of values in `pots` that doesn't include two adjacent pot values. *Hint*: `sum([])` is 0, `sum([2])` is 2, and `max(3, 4)` is 4.

```
def leap(pots):
    """Return the maximal value of collecting pots that are not adjacent.

    >>> leap([2, 4, 3]) # Collect 2 and 3
    5
    >>> leap([4, 20, 9, 3, 6, 2]) # Collect 20 and 6
    26
    """
    if len(pots) <= 1:

        return sum(pots)

    return max(pots[0] + leap(pots[2:]), leap(pots[1:]))
```

(b) (6 pt) The *Leap*rechaun Ida Clare declares, "Let's use a declarative language!" The `pots` table contains indexed pot values. A path is a comma-separated list of `value`s from `pots`, ordered by their `place`s.

****A path cannot contain values from adjacent places.****

Implement a `select` statement that returns a one-row, two-column table. The first column should contain the maximal path through `pots`. The second column should contain the total value of the maximal path. Your solution should provide the correct result even if the contents of the `pots` table were to change.

```
create table pots as
  select 0 as place,  4 as value union
  select 1         , 20          union
  select 2         ,  9          union
  select 3         ,  3          union
  select 4         ,  6          union
  select 5         ,  2;

with
  paths(path, last, total) as (

    select value, place, value from pots union

    select path || ", " || value, place, total + value

        from paths, pots

        where place - last > 1
  )

select path, max(total) from paths;

-- Expected result:
--   20,6|26
```

(c) (4 pt) Higher-order *Leap*rechauns can collect any pots they wish, but get a bonus when they jump over some pots. They always take the first pot before jumping. Implement `gather`, which takes a list of `pots` and a function `bonus(k)` that returns the bonus for jumping over exactly `k` pots. `gather` returns the maximum total gold plus bonus that is possible to attain, along with a linked list of the pot values collected. The `Link` class appears on the Midterm 2 Study Guide.

```
def gather(pots, bonus):
    """Return the maximum total value of pots gathered *plus* jump bonuses.
    Also return a linked list of pot values gathered to reach this total.

    >>> gold = [4, 20, 9, 3, 6, 2]
    >>> gather(gold, lambda k: 0) # No jumping bonus, so gather everything
    (44, Link(4, Link(20, Link(9, Link(3, Link(6, Link(2)))))))

    >>> hop = lambda k: [0, 10, 0, 0, 0, 0, 0, 0][k]
    >>> gather(gold, hop) # Jump 0, Jump 0, Jump 1 (+10), Jump 1 (+10) to end
    (59, Link(4, Link(20, Link(9, Link(6)))))

    >>> leap = lambda k: [0, 0, 30, 20, 0, 0, 0, 0][k]
    >>> gather(gold, leap) # Jump 2 (+30), Jump 2 to end (+30)
    (67, Link(4, Link(3)))
    """


    if pots == []:


        return 0, Link.empty


    def total(k):


        return pots[0] + bonus(k) + gather(pots[k+1:], bonus)[0]


    best = max(range(len(pots)), key=total)


    rest = gather(pots[best+1:], bonus)[1]


    return total(best), Link(pots[0], rest)
```

6. (10 points) SQL in Scheme

A SQL-like table can be implemented in Scheme as a list of lists. The first element of a table is a list of column names. Each subsequent element is a list of values in a row. (*Note*: The semicolon line is just a comment.)

```
scm> (define cafes (list
       '(cafe    item    )
       ; ------  --------
       '(nefeli  espresso)
       '(nefeli  bagels  )
       '(fsm     coffee  )
       '(fsm     bagels  )
       '(fsm     espresso)))
```

(a) **(2 pt)** The `get` procedure takes a symbol `column`, a list of `columns` and a list of `values`. It returns the value at the same index that `column` appears in `columns`. Assume that `column` appears within `columns`.

```
scm> (get 'cafe '(cafe item) '(nefeli  bagels))
nefeli
scm> (get 'item '(cafe item) '(nefeli  bagels))
bagels
```

Cross out whole lines below so that `get` is implemented to match the examples above.

```
; Get an element of values by matching column to an element of columns
(define (get column columns values)
  (if (equal? column (car columns))
      (car values)
      (get column (cdr columns) (cdr values))))
```

(b) **(2 pt)** The `insert` procedure takes a `key`, a `value`, and a list of `groups`. A group is a list that begins with a key, followed by values. The `insert` procedure returns an updated list of groups with `value` added to the beginning of the group for `key`. If `key` is not in `groups`, then a new group is added to the end.

```
scm> (insert 'b 5 nil)
((b 5))
scm> (insert 'a 1 (insert 'b 2 (insert 'a 3 (insert 'c 4 (insert 'b 5 nil)))))
((b 2 5) (c 4) (a 1 3))
```

Cross out whole lines below so that `insert` is implemented to match the examples above.

```
; Insert value into the group within groups that starts with key
(define (insert key value groups)
  (if (null? groups)
      (list (list key value))
      (if (equal? key (car (car groups)))
          (cons (cons key (cons value (cdr (car groups))))
                (cdr groups))
          (cons (car groups)
                (insert key value (cdr groups))))))
```

The following procedure is implemented correctly for you as a reference.

```
(define (map fn s) (if (null? s) s (cons (fn (car s)) (map fn (cdr s)))))
```

(c) **(3 pt)** Implement `from`, a procedure that takes a table (e.g., `cafes`) and returns a list of row procedures. A row procedure represents a row; it takes a column name and returns the value for that column in the row.

```
scm> (define first-row (car (from cafes))) ; first-row is a procedure
first-row
scm> (first-row 'cafe)
nefeli
scm> (first-row 'item)
espresso
```

You may use `get`, `map`, and list manipulation procedures such as `car`, `cdr`, `cons`, and `list`. **You may only use `lambda` and call expressions. No other special forms (such as `if`) are allowed.**

```
(define (from table)
  (map (lambda (row)
         (lambda (column)
           (get column (car table) row)))
       (cdr table)))
```

The following procedure is implemented correctly for you, but you must discover its behavior and purpose.

```
(define (group-by column)
  ; Hint: rows is a list of procedures that are returned by "from"
  (define (group-all rows groups)
    (if (null? rows) groups
        (group-all (cdr rows) (insert ((car rows) column) (car rows) groups))))
  group-all)
```

(d) **(3 pt)** Implement `select`, which takes a `column` name and the results of calling `from` and `group-by`. It returns a list of two-element lists, which each contain a group key and a list of column values.

```
scm> (select 'item (from cafes) (group-by 'cafe))
((nefeli (bagels espresso)) (fsm (espresso bagels coffee)))

scm> (select 'cafe (from cafes) (group-by 'item))
((espresso (fsm nefeli)) (bagels (fsm nefeli)) (coffee (fsm)))
```

You may use the `map` procedure, as well as list manipulation procedures such as `car`, `cdr`, `cons`, and `list`. **You may only use `lambda` and call expressions. No other special forms (such as `if`) are allowed.**

```
(define (select column rows grouping)
  (map (lambda (group)
         (list (car group)
               (map (lambda (row) (row column))
                    (cdr group))))
       (grouping rows '())))
```