# CS 61A Structure and Interpretation of Computer Programs
# Spring 2018

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`, and `abs`.
- You **may not** use example functions defined on your study guides unless clearly specified by the question.
- 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) The Floss** *(At least one of these is out of Scope: OOP, WWPD, Lambda, Python Lists, Mutation)*

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.

```
class Forth:
    next = 1

    def __init__(self, k):
        self.k = k

    def __repr__(self):
        return str(self.k) + '*'

    def go(self, k):
        if k == 1:
            print(self)
        if k:
            return self.next.go(k-1)
        print(self)

class Back(Forth):
    @property
    def next(self):
        return self

g = [Forth(n) for n in range(3, 8)]
for i in range(4):
    g[i].next = g[i+1]
g[3] = Back(2)

m = map(lambda o: o.k, g)
```

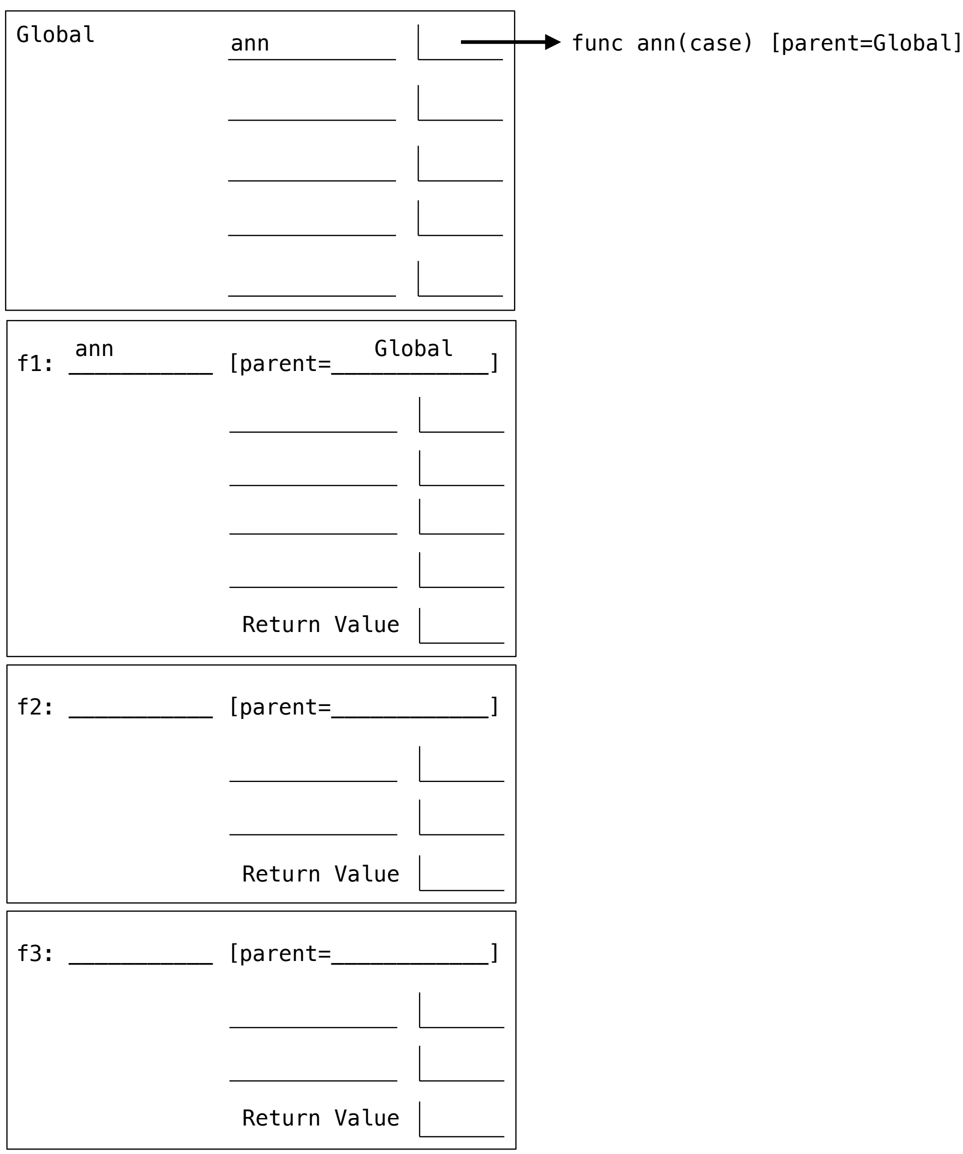
| Expression | Output |
|---|---|
| `g[0]` | 3* |
| `[next(m), next(m)]` | |
| `next(next(iter([[0]])))` | |
| `len([map(print, g)])` | |
| `g[0].go(2)` | |
| `g[0].go(4)` | |
| `[x.next for x in g[3:]]` | |

**2. (8 points) Announcements** *(All are in Scope: Lambda, Python Lists, Mutation, Nonlocal, HOFs)*

```
def ann(case):
    def the(s):
        nonlocal case
        case = s[:]
    it = list(ounce)
    it.extend(case.pop())
    if it is not the(case):
        return lambda t: [2]+s+[t]
    else:
        return lambda u: [2]+s+[u]
ounce, ments = [2, [0, 1]], 8
s = ounce[1]
ann(ounce)(ments)
```

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.**

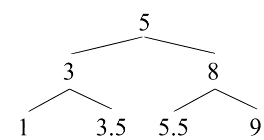


**3. (7 points) Binary Trees** *(All are in Scope: Tree Class, Recursion)*

**Definition**. A *binary search tree* is a `BTree` instance for which the label of each node is larger than all labels in its left branch and smaller than all labels in its right branch.

**(a) (5 pt)** Implement `largest`, which takes a binary search tree `t` and a number `x`. It returns the largest label in `t` that is smaller than `x`. If no such label exists, it returns 0. **Assume that `t` contains only positive numbers as labels.** The `BTree` class is on page 2 (bottom of left column) of the Midterm 2 Study Guide.

```
def largest(t, x):
    """Return the largest label in t that is less than x, or 0 if none exists.

    >>> a = BTree(5, BTree(3, BTree(1), BTree(3.5)), BTree(8, BTree(5.5), BTree(9)))
    >>> largest(a, 5)
    3.5
    >>> largest(a, 5.1)
    5
    >>> largest(a, 6)
    5.5
    >>> largest(a.right, 5)
    0
    """
    if t is BTree.empty:

        return 0


    elif ______________________________________________________________________:


        return largest(______________________________, ______________________________)


    y = largest(_________________________________, _________________________________)

    if y:

        return ________________________________________________________________


    return ____________________________________________________________________
```




**(b) (2 pt)** Implement `second`, which takes a binary search tree `t` **containing only positive numbers**, and a number `x`. It returns the **second largest** label in `t` that is smaller than `x`.

```
def second(t, x):
    """Return the second largest label in t that is less than x, or 0 if none exists.

    >>> a = BTree(5, BTree(3, BTree(1), BTree(3.5)), BTree(8, BTree(5.5), BTree(9)))
    >>> second(a, 5)
    3
    >>> second(a, 5.1)
    3.5
    >>> second(a.left, 2)
    0
    """
    return ___________________________________________________________
```

## 4. (10 points) Apply Yourself

**(a) (6 pt)** *(All are in Scope: HOFs, Generators, Recursion)* Implement `times`, which takes a one-argument function `f` and a starting value `x`. It returns a function `g` that takes a value `y` and returns the minimum number of times that `f` must be called on `x` to return `y`. Assume that calling `f` repeatedly on `x` eventually results in `y`.

```
def times(f, x):
    """Return a function g(y) that returns the number of f's in f(f(...(f(x)))) == y.

    >>> times(lambda a: a + 2, 0)(10)   # 5 times: 0 + 2 + 2 + 2 + 2 + 2 == 10
    5
    >>> times(lambda a: a * a, 2)(256)  # 3 times: square(square(square(2))) == 256
    3
    """
    def repeat(z):
        """Yield an infinite sequence of z, f(z), f(f(z)), f(f(f(z))), f(f(f(f(z)))), ...."""

        yield ______________________________________________________________________________

        ________________________________________________________________________________


    def g(y):

        n = 0

        for w in repeat(_____________________________________________________________):

            if ______________________________________________________________________:

                ______________________________________________________________________

            __________________________________________________________________________

    return g
```

**(b) (2 pt)** *(At least one of these is out of Scope: Orders of Growth)* Circle the $\Theta$ expression that describes how many steps are required to evaluate `f(f(n))`, assuming `f(n)` returns $2^n$ for all $n$, and $\Theta(n)$ steps are required to evaluate `f(n)`.

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

**(c) (2 pt)** *(At least one of these is out of Scope: Orders of Growth)* Circle the $\Theta$ expression that describes how many steps are required to evaluate `g(g(n))`, assuming `g(n)` returns $\sqrt{n}$ for all $n$, and $\Theta(n)$ steps are required to evaluate `g(n)`.

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

**5. (12 points) Functions As Expected** *(All are in Scope: HOFs, Lambda, Python Lists, Recursion, Tree Recursion)*

**Definition**. For $n > 1$, an *order n function* takes one argument and returns an order $n-1$ function. An order 1 function is any function that takes one argument.

**(a) (6 pt)** Implement `scurry`, which takes a function `f` and a positive integer `n`. `f` must be a function that takes a list as its argument. `scurry` returns an order $n$ function that, when called successively $n$ times on a sequence of values $x_1, x_2, \ldots x_n$, returns the result of calling `f` on a list containing $x_1, x_2, \ldots x_n$.

```
def scurry(f, n):
    """Return a function that calls f on a list of arguments after being called n times.

    >>> scurry(sum, 4)(1)(1)(3)(2)  # equivalent to sum([1, 1, 3, 2])
    7
    >>> scurry(len, 3)(7)([8])(-9)  # equivalent to len([7, [8], -9])
    3
    """
    def h(k, args_so_far):

        if k == 0:

            return ________________________________________________________________________

        return ____________________________________________________________________________

    return ________________________________________________________________________________
```

**(b) (6 pt)** Implement `factorize`, which takes two integers `n` and `k`, both larger than 1. It returns the number of ways that `n` can be expressed as a product of non-decreasing integers greater than or equal to `k`.

```
def factorize(n, k=2):
    """Return the number of ways to factorize positive integer n.

    >>> factorize(7)  # 7
    1
    >>> factorize(12) # 2*2*3, 2*6, 3*4, 12
    4
    >>> factorize(36) # 2*2*3*3, 2*2*9, 2*3*6, 2*18, 3*3*4, 3*12, 4*9, 6*6, 36
    9
    """
    if ____________________________________________________________________________________:

        return 1

    elif __________________________________________________________________________________:

        return 0

    elif __________________________________________________________________________________:


        return factorize(________________________________, ________________________________)


    return ________________________________________________________________________________
```

**6. (16 points) Scheme Forever**

**(a) (4 pt)** *(At least one of these is out of Scope: Scheme, Tail Recursion)* Implement `fibs`, which takes a positive integer `n` and prints out the first `n` Fibonacci numbers in order, one on each line. For example, `(fibs 7)` prints 0 on one line, 1 on the next, then 1, 2, 3, 5, and 8; seven lines in total. **Your implementation must run in constant space to receive full credit.**

```
(define (fibs n)
  ; Print the first n Fibonacci numbers. Each one is the sum of the previous two.
  ; (fibs 7) prints the Fibonacci numbers 0 1 1 2 3 5 8 each on a different line.

  (define (taipei a b k)

    (if (= k n)

        (print ________________________________________________________________________)

        ________________________________________________________________________________))

  (taipei 1 0 1))
```

**(b) (4 pt)** *(At least one of these is out of Scope: Scheme Streams)* Write the first 7 elements of each stream that results from the two calls to `e` below. *Note*: In Scheme, `quotient` performs floor division like `//` in Python, and `remainder` is like `%` in Python.

```
(define (e n d) (cons-stream (quotient n d) (e (* 10 (remainder n d)) d)))

(e 1 8)   ; Starts with: ______   ______   ______   ______   ______   ______   ______

(e 2 3)   ; Starts with: ______   ______   ______   ______   ______   ______   ______
```

**(c) (4 pt)** *(At least one of these is out of Scope: Scheme Macros)* Implement `lambda-macro`, a macro that creates anonymous macros. A `lambda-macro` expression has a list of formal parameters and one body expression. It creates a macro with those formal parameters and that body. Assume that the symbol `anon` is not use anywhere else in a program that contains `lambda-macro`.

```
(define-macro (lambda-macro bindings body)
    ; A lambda-macro expression evaluates to a macro.
    ; For example: ((lambda-macro (expr) (car expr)) (+ 1 2)) evaluates to the symbol +

    `(begin (_______________________ ___________________________ __________________________)

            anon))
```

**(d) (4 pt)** *(At least one of these is out of Scope: Scheme Malformed List)* Implement `dotted?`, which takes a value `s`. It returns whether `s` is a dotted list or contains a dotted list anywhere within it.

```
(define (dotted? s)
  ; Return whether s contains a dotted list.
  ; Examples that are dotted:     (1 2 . 3) , (1 (2 . 3)) , (((1 . 2)) 3)
  ; Examples that are not dotted: (1 2 3)   , (1 2.3)     , ((1) ((2)) 3)

  (cond (( ___________________________________ ( ___________________________________ s)) #f)

        ((not-pair-or-nil? ___________________________________________________________ ) #t)

        (else ___________________________________________________________________________)))

(define (not-pair-or-nil? s) (and (not (pair? s)) (not (null? s))))
```

**7. (10 points) Gotta Select 'Em All** *(All are in Scope: SQL, SQL Aggregation)*

For the questions below, assume that the following two SQL statements have been executed. The `pokedex` table describes the names of some Pokémon and their heights in inches. The `evolve` table describes how those Pokémon can evolve into the other Pokémon in the `pokedex`.

```
CREATE TABLE pokedex AS
  SELECT "Eevee" AS name, 12 as height UNION
  SELECT "Jolteon"      , 31           UNION
  SELECT "Leafeon"      , 39           UNION
  SELECT "Bulbasaur"    , 28           UNION
  SELECT "Ivysaur"      , 39           UNION
  SELECT "Venasaur"     , 79           UNION
  SELECT "Charmander"   , 24           UNION
  SELECT "Charmeleon"   , 43           UNION
  SELECT "Charizard"    , 67;
```

```
CREATE TABLE evolve AS
  SELECT "Eevee" AS before, "Jolteon" AS after UNION
  SELECT "Eevee"          , "Leafeon"          UNION
  SELECT "Bulbasaur"      , "Ivysaur"          UNION
  SELECT "Ivysaur"        , "Venasaur"         UNION
  SELECT "Charmander"     , "Charmeleon"       UNION
  SELECT "Charmeleon"     , "Charizard";
```

(a) **(4 pt)** Write a SQL statement that adds a new row to the `evolve` table for each pair of Pokémon for which `before` evolves to `after` in two steps. For example, Charmander can evolve twice: first to Charmeleon and then to Charizard. Therefore, `("Charmander", "Charizard")` should be added as a row. Likewise, `("Bulbasaur", "Venasaur")` should also be added. Your statement should behave correctly even if the rows in `evolve` and `pokedex` were different. The rows can be added in any order.

```
________________________________________________________________________________

SELECT _________________________________________________________________________

FROM ___________________________________________________________________________

WHERE __________________________________________________________________________
```

(b) **(6 pt)** Write a SELECT statement that results in a table with one row for each Pokémon that can evolve. The table should have two columns: the first contains the name of the Pokémon that can evolve, and the second contains the maximum increase in height that it can attained by evolving. For example, Eevee can grow as much as 27 inches (when evolving to Leafeon), so the result should contain the row `("Eevee", 27)`. Your statement should behave correctly even if the rows in `evolve` and `pokedex` were different. The result should only consider ways of evolving that are described by a single row in the `evolve` table.

```
SELECT _________________________________________________________________________

FROM ___________________________________________________________________________

WHERE __________________________________________________________________________

________________________________________________________________________________;
```