CS 61A Structure and Interpretation of Computer Programs
Spring 2016

TEST 1

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is open book, open notes, closed computer, closed calculator. The official CS 61A midterm 1 study guide will be provided.
- 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	
Name of the person to your left	
Name of the person to your right	
I pledge my honor that during this examination I have neither given nor received assistance. **(please sign)**	

1. (12 points) Evaluate This!

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`". If an expression yields (or prints) a function, write "`<Function>`". No answer requires more than 3 lines. (It's possible that all of them require even fewer.) The first two rows have been provided as examples.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`, plus all values passed to `print`.

(a) (10 pt) *(At least one of these is out of Scope: WWPD, Lambda, Lists)* Assume that `python3` has executed the statements on the left:

```
y = 7

def b(x):
  return lambda y: x(y)

def c(x):
  return 3

w = b(c)

def c(x):
  return x
```

Expression	Interactive Output
`pow(2, 3)`	8
`print(4, 5) + 1`	4 5 Error
`(3 and abs)(-1)`	
`print(3) or 1/0`	
`print`	
`([1, 2, 3] if y // (y+1) else [4, 5])[1]`	
`w(5)`	

(b) (2 pt) *(At least one of these is out of Scope: WWPD, HOFs, Lists)* Assume that `python3` has executed the following statements:

```
def d(S):
    def f(k):
        return k < len(S) and (S[k] == S[0] or f(k+1))
    if len(S) == 0:
        return False
    else:
        return f(1) or d(S[1:])
```

Expression	Interactive Output
`print(d([1, 2, 3]), d([0, 1, 2, 1, 0]))`	

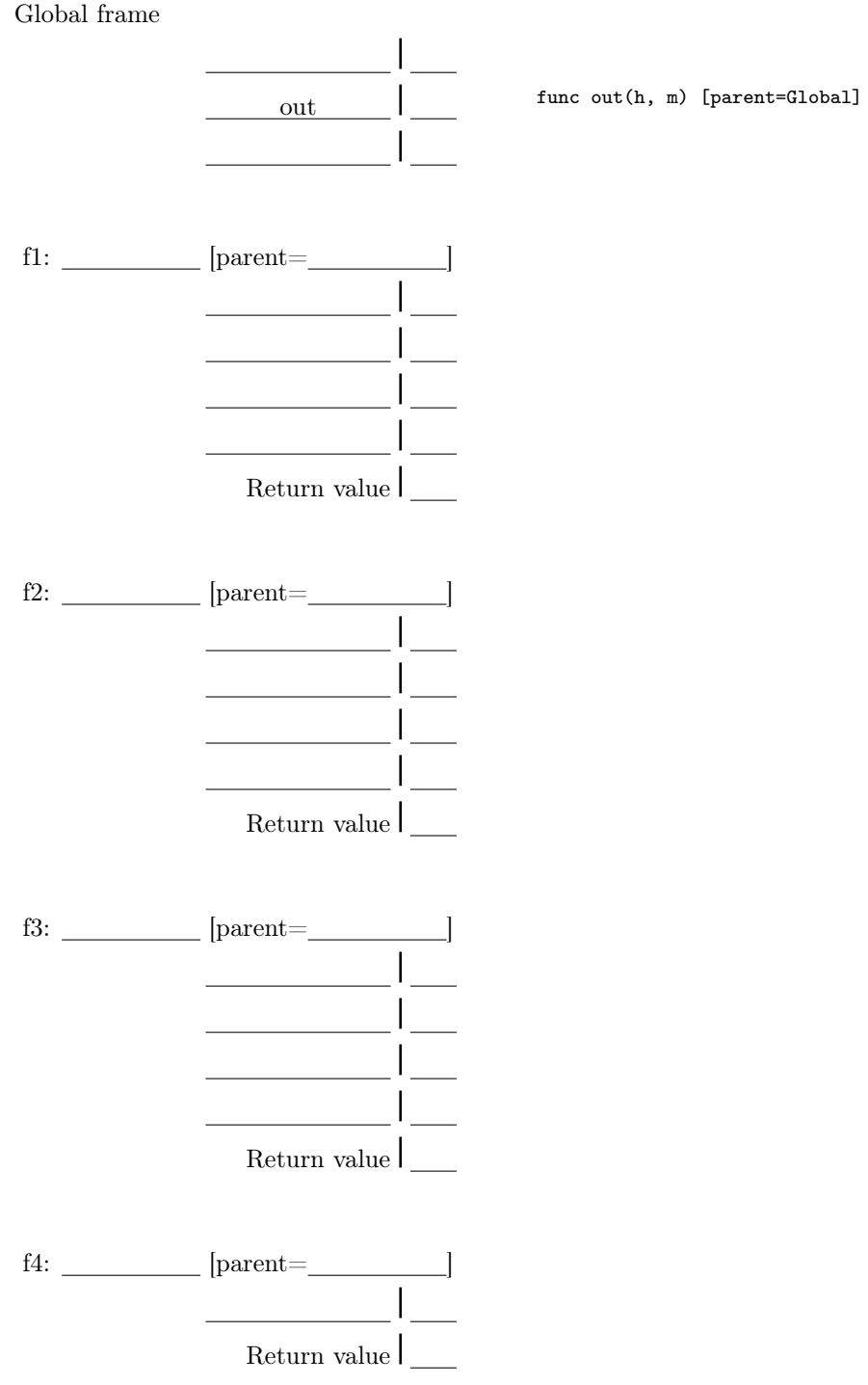
2. (12 points) Environmental Policy

(a) (6 pt) *(At least one of these is out of Scope: Environment Diagrams, HOFs, Recursion)* 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 or referenced during execution.
- Show the return value for each local frame.

```
1  y = 3
2  def out(h, m):
3      y = 5 * m
4      def inner():
5          return y
6      if m == 0:
7          return h
8      else:
9          return out(inner, m-1)
10 v = out(None, 1)()
```



(b) (6 pt) *(At least one of these is out of Scope: Environment Diagrams, Self-Reference, Lambda)* 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.* The `<line ...>` annotation in a lambda value gives the line in the Python source of a lambda expression.

A complete answer will:

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

```
1 def lazy(n):
2     return lambda k: (n if k==0 else lazy(n+1))
3 v = lazy(4)(1)(0)
```

Global frame

lazy | ___ `func lazy(n) [parent=Global]`

_____ | ___

f1: _____ [parent=_____]

_____ | ___

_____ | ___

Return value | ___

f2: _____ [parent=_____]

_____ | ___

_____ | ___

Return value | ___

f3: _____ [parent=_____]

_____ | ___

_____ | ___

Return value | ___

f4: _____ [parent=_____]

_____ | ___

_____ | ___

Return value | ___

3. (1 points) Extra Who described Italy (in translation) as "the land where lemon blossoms blow, / And through dark leaves the golden oranges glow."?

4. (4 points) Going Around in Cycles *(All are in Scope: Lists, Control)*

Consider a list of integers, `L`, in which each integer is a value in 0 to `len(L)-1` (inclusive). That is, each item in the list could also be used as a valid index into the list. We'll say that `L` has a *cycle of length* k iff for some sequence of index values $i_1, i_2, \ldots, i_k$, we have `L[`i_1`] = `i_2`, L[`i_2`] = `i_3`, ..., L[`i_k`] = `i_1. For example, the list `L = [3, 1, 4, 5, 0, 2]` has a cycle of length 1: $1 \to 1$, and a cycle of length 5: $0 \to 3 \to 5 \to 2 \to 4 \to 0$. For the purposes of this problem, we'll count something as a cycle of length k only if the first k indices in it contain no repetitions (so `L` in the example does not contain a cycle of length 2 even though $1 \to 1 \to 1$.

Write a function `has_cycle(L, k)` that returns a true value iff `L` has a cycle of length k by filling in the template below.

```
def has_cycle(L, k):
    """True iff L has a cycle of length K>=1.  Assumes 0 <= L[i] < len(L)
    for 0 <= i < len(L).
    """
    def cycle_at(s):
        """L has a cycle of length K starting at S."""

        ______________________________________________

        n = 1

        while ________________________________________:


            if ________________________________________:

                return False


            ______________________________________________

            n += 1


        return ________________________________________

    for j in range(len(L)):


        if ________________________________________:


            return ____________________________________


    return ____________________________________________
```

5. (4 points) Your Father's Parentheses *(All are in Scope: Tree Recursion)*

Suppose we have a sequence of quantities that we want to multiply together, but can only multiply two at a time. We can express the various ways of doing so by counting the number of different ways to parenthesize the sequence. For example, here are the possibilities for products of 1, 2, 3, 4 and 5 elements:

Product	a	ab	abc	abcd	abcde		
Count	1	1	2	5	14		
Parenthesizations	a	ab	a(bc) (ab)c	a(b(cd)) a((bc)d) (ab)(cd) (a(bc))d ((ab)c)d	a(b(c(de))) a(b((cd)e)) a((bc)(de)) a((b(cd))e) a(((bc)d)e)	(ab)(c(de)) (ab)((cd)e) (a(bc))(de) ((ab)c)(de) (a(b(cd)))e	(a((bc)d))e ((ab)(cd))e ((a(bc))d)e (((ab)c)d)e

Assume, as in the table above, that we don't want to reorder elements.

Define a function `count_groupings` that takes a positive integer n and returns the number of ways of parenthesizing the product of n numbers. (You might not need to use all lines.)

```
def count_groupings(n):
    """For N >= 1, the number of distinct parenthesizations
    of a product of N items.
    >>> count_groupings(1)
    1
    >>> count_groupings(2)
    1
    >>> count_groupings(3)
    2
    >>> count_groupings(4)
    5
    >>> count_groupings(5)
    14
    """
    if n == 1:

        return ____________________________________________________________


    ____________________________________________________________________


    i = ________________________________________________________________


    while ____________________________________________________:


        ____________________________________________________________________

        i += 1

    return ____________________________________________________________
```

6. (8 points) Amazing *(All are in Scope: Recursion, HOFs)*

In lecture, we did some problems involving finding one's way through a maze, representing the maze as a predicate (boolean function)—the parameter `blocked`. The idea was that `blocked(x, y)` iff the grid square at row `y` and column `x` was blocked. Let's consider a different representation.

Again, mazes will be represented by functions, but with a different specification. A maze function, say `M`, in this formulation describes the state of the maze from the point of view of a maze runner. `M` accepts a single argument, `direction`, which describes the action the runner tries to take: either the string "south" or "west", indicating that the runner tries to take one step south or one step west, respectively. `M` then returns either:

- Another maze function that represents the state of the runner after taking the indicated step. It again accepts a direction and returns one of these three things;
- The string "exit", signifying that the move causes the runner to successfully reach the exit; or
- The string "dead end", which indicates that the runner was unable to take the requested action because of being blocked by a wall.

(a) (4 pt) Define a function `pred_maze` that returns a maze function (as described above).
If `M = pred_maze`(x, y, P, e), then `M` represents a runner in a maze in which

- The runner is at coordinates (x, y).
- Squares (a, b) for which $a \leq e$ are unblocked and are exit squares.
- A square at (a, b) is otherwise open iff $P(a, b)$.

For example, suppose that `Q` is a function such that `Q(a,b)` is true on all the white squares in the following grid (where, just to be different, the lower-left corner represents position (-5, -5)):

Then we'd have

```
>>> M0 = pred_maze(0, 1, Q, -4)
>>> M1 = M0('west')
>>> M1
<function ...>
>>> M1('west')
'dead end'
>>> M1('south')
<function ...>
>>> M2 = pred_maze(-3, -1, Q, -4)
>>> M2('west')
'exit'
```

Fill in your solution on next page.

```
def pred_maze(x0, y0, open, exit):
    """Return a maze in which the runner is at (X0, Y0), every square
    (a, b) where a <= EXIT is an exit, and otherwise a square (a, b)
    is open iff OPEN(a, b).  It is assumed that (X0, Y0) is open."""

    def maze(____________________________):

        x, y = (x0, y0 - 1) if dir == "south" else (x0 - 1, y0)


        if ____________________________________________:


            return ______________________________________


        elif ___________________________________________:


            return ______________________________________

        else:

            return ______________________________________


    return maze
```

(b) (4 pt) Define a function `path_out` that takes a maze and returns a string describing a path to an exit, or returns `None` iff there is no such path. A call `path_out(M)` will return a string such as

```
'south west south west west'
```

(as in the doctest below) to indicate that the necessary path takes one step south, then a step west, then south, and then two steps west. (Don't worry about extra spaces in the returned string.) When multiple paths exist, return any one. Using the previous value of function `Q`, for example, we'd see:

```
>>> M = pred_maze(-1, 1, Q, -4)
>>> path_out(M)
'south west south west west'
>>> M = pred_maze(2, -1, Q, -4)
>>> path_out(M)   # Returns None
```

FYI: The '+' operator on strings is string concatenation ('+=') also works.)

```
def path_out(M):
    """Given a maze function M in which the runner is not yet out
    of the maze, returns a string of the form "D1 D2 ...", where each
    Di is either south or west, indicating a path from M to an exit,
    or None iff there is no such path."""

    for dir in ["south", "west"]:


        next = ________________________________________


        if ____________________________________________:


            return _____________________________________


        elif ___________________________________________:


            rest_of_path = _______________________________


            if _____________________________________________:


                return ___________________________________

    return None
```