CS 61A Structure and Interpretation of Computer Programs
Summer 2014

Midterm 1 Solutions

INSTRUCTIONS

- You have 2 hours to complete the exam.
- The exam is closed book, closed notes, and closed electronics, except one hand-written 8.5” × 11” cheat sheet of your own creation, and The Environment Diagram Rules.
- Mark your answers ON THE EXAM ITSELF. Answers outside of the space allotted to problems will *not* be graded. If you are not sure of your answer you may wish to provide a *brief* explanation.

Full 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)**	

1. **(1 points)** **Your thoughts?** What makes you happy? (Alternatively, draw us a nice doodle). You can also take this opportunity to give us feedback.

2. (8 points) What will Python output?

Include all lines that the interpreter would display. If it would display a function, then write Function. If it would cause an error, write Error. Assume that you have started Python 3 and executed the following. **These are entered into Python exactly as written.**

```
def welcome():
    if a == 0:
        return 'hello, welcome to your exam'
    return 'prepare for tricks.'

def last_night(n):
    for i in range(n):
        return 'exams'

pi = [3, 1, 4, 1, 5, 9, 2, 6, 5, 4]
cut = lambda thing: thing[2:]
slice_of = lambda thing: thing[2:8:2]
def mystery(x):
    if x and (x + 1):
        return 'mystery'
    return mystery
```

Expression	Interactive Output
`4`	4
`print(5)`	5
`welcome()`	Error
`last_night(308)`	'exams'
`(lambda x, y: x + y(x))(4, lambda y: 5)`	9
`[3 for x in range(30) if x > 26]`	[3, 3, 3]
`cut(slice_of(pi))`	[2]
`cut(mystery(-1)(20))`	'stery'
`cut(mystery(20)(-1))`	Error
`print(mystery(print(20)))`	20 Function

3. (12 points) Environment Diagrams

(a) (6 pt) Environmental, my dear Watson

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.* You may want to keep track of the stack on the left, but this is not required.

A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.
- The first function created by `lambda` should be labelled λ_1, the next one should be λ_2, and so on.

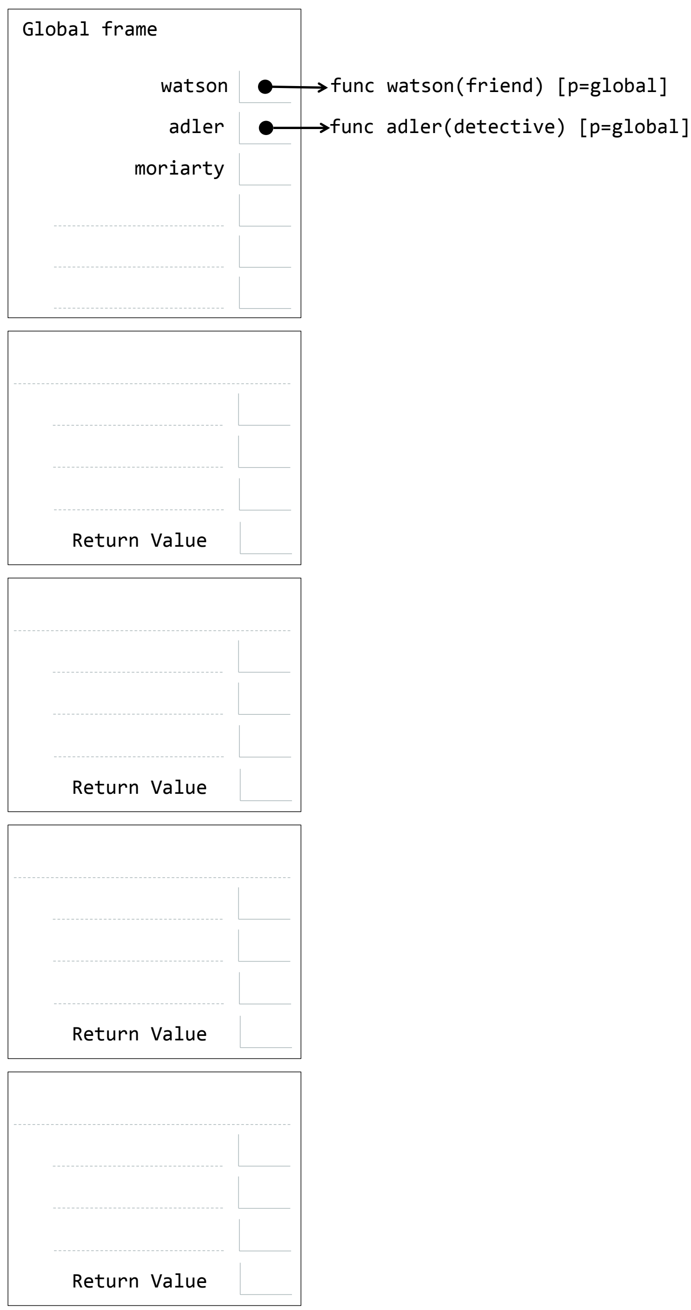
```
def watson(friend):
    trick = 5
    def holmes():
        return friend - 10
    return holmes

def adler(detective):
    trick = 4
    return detective(trick)

moriarty = adler(watson)
moriarty()
```

Stack

global



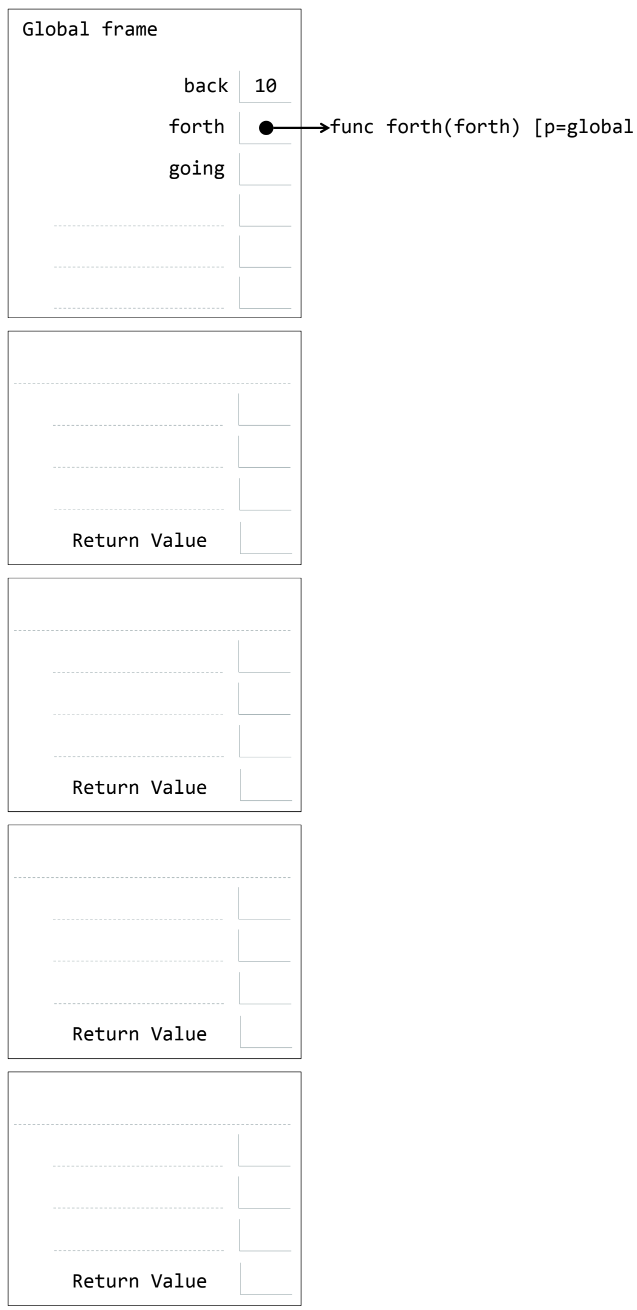
(b) (6 pt) Well... that escalated quickly

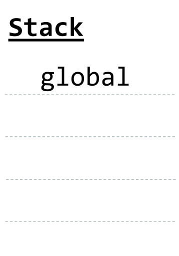
Note: This is a hard question. Fill in the environment diagram that results from executing the code below until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.* You may want to keep track of the stack on the left, but this is not required. You should be extra careful here. Hint: What is the operator? What is the operand?

A complete answer will:

- Add all missing names, labels, and parent annotations to all local frames.
- Add all missing values created during execution.
- Show the return value for each local frame.
- The first function created by `lambda` should be labelled λ_1, the next one should be λ_2, and so on.

```
back = 10
def forth(forth):
    back = lambda back: forth // back(15)
    return back
going = forth(back)(lambda forth: 5)
```





4. (5 points) Here We Go Again

Define a function `wheres_waldo`, which takes in a linked list which may or may not contain the string `'Waldo'` as an element, and returns the index of `'Waldo'` if it exists somewhere in the list, and `'Nowhere'` if it does not. *Do not assume we have* `get_item` *defined.* Note that `linked_list` is *not* a deep linked list.

```
def wheres_waldo(linked_list):
    """
    >>> lst = link("Moe", link("Larry", link("Waldo", link("Curly", empty))))
    >>> wheres_waldo(lst)
    2
    >>> wheres_waldo(link(1, link(2, empty)))
    'Nowhere'
    """

    if linked_list == empty:
        return 'Nowhere'
    elif first(linked_list) == 'Waldo':
        return 0
    found_him = wheres_waldo(rest(linked_list))
    if found_him == 'Nowhere':
        return found_him
    return 1 + found_him
```

5. (12 points) Piled Higher and Deeper

(a) (4 pt) Higher List Magic Write the function `inhexing`, which takes in a Python list of numbers `lst`, a function `hex`, and an integer `n`, and returns a new list where every n^{th} element is replaced by the result of calling `hex` on that element.

```
def inhexing(lst, hex, n):
    """
    >>> inhexing([1, 2, 3, 4, 5], lambda x: 'Poof!', 2)
    [1, 'Poof!', 3, 'Poof!', 5]
    >>> inhexing([2, 3, 4, 5, 6, 7, 8], lambda x: x + 10, 3)
    [2, 3, 14, 5, 6, 17, 8]
    """

    result = []
    for i in range(len(lst)):
        if (i + 1) % n == 0:
            result += [ hex(lst[i]) ]
        else:
            result += [ lst[i] ]
    return result
```

(b) (8 pt) Deeper List Magic Now write `deep_inhexing`, for deep *Python* lists. It takes in a DEEP Python list, a function, and a number. It returns a new list where every n^{th} element is replaced by the function applied to that element. If it encounters a list as an element, it recurses on the sublist, resetting the counter, even if the sublist was an `n`th element. Recall you can use the expression `type(x) == type([])` to test if `x` is a Python list. **Make sure you read and understand all the doctests!**

```
def deep_inhexing(lst, hex, n):
    """
    >>> deep_inhexing([1, 2, 3, 4, 5, 6], lambda x: x + 10, 3)
    [1, 2, 13, 4, 5, 16]
    >>> deep_inhexing([1, [[2]], [3, 4, [5]]], lambda x: 'Poof!', 1)
    ['Poof!', [['Poof!']], ['Poof!', 'Poof!', ['Poof!']]]
    >>> deep_inhexing([1, [2], 3], lambda x: 'Poof!', 2)
    [1, [2], 3]
    >>> deep_inhexing([1, [2, 3], 4, [5, 6]], lambda x: 'Poof!', 2)
    [1, [2, 'Poof!'], 4, [5, 'Poof!']]
    >>> deep_inhexing([[2, 3], 4, [5, 6], [7]], lambda x: 'Poof!', 2)
    [[2, 'Poof!'], 'Poof!', [5, 'Poof!'], [7]]
    >>> deep_inhexing([2, [4, [6, [8, 10]]]], lambda x: 'Poof!', 2)
    [2, [4, [6, [8, 'Poof!']]]]
    """

    def helper(lst, counter):
        if lst == []:
            return []
        first, rest = lst[0], lst[1:]
        if type(first) == type([]):
            return [helper(first, 1)] + helper(rest, counter + 1)
        elif counter % n  == 0:
            return [hex(first)] + helper(rest, counter + 1)
        else:
            return [first] + helper(rest, counter + 1)
    return helper(lst, 1)
```

6. (2 points) Data Abstraction

True or False: Code that uses ADTs may behave as normal when you commit a Data Abstraction Violation. If True, explain why we care about ADTs. If False, explain what would break.

The statement is (write True/False): True

Explanation: We use ADTs because they help us separate the problem of how to represent data from the problem of how to use that data. This separation allows us to write cleaner, more maintainable code, which is easier to modify. For example, we can change just the constructors and the selectors to change the representation, and all the other code that uses the data should just work.

7. (5 points) Recursion on Tree ADT

Define a function `dejavu`, which takes in a tree of numbers `t` and a number `n`. It returns `True` if there is a path from the root to a leaf such that the sum of the numbers along that path is `n` and `False` otherwise. Reminder: The constructor and selectors are `tree`, `datum` and `children`.

```
def dejavu(t, n):
    """
    >>> my_tree = tree(2, [tree(3, [tree(5), tree(7)]), tree(4)])
    >>> dejavu(my_tree, 12) # 2 -> 3 -> 7
    True
    >>> dejavu(my_tree, 5)  # Sums of partial paths like 2 -> 3 don't count
    False
    """

    if children(t) == []:
        return n == datum(t)
    for child in children(t):
        if dejavu(child, n - datum(t)):
            return True
    return False
```

8. (3 points) Orders of Growth

(a) (1 pt) Consider the following function definition:

```
def foo(n):
    times_table = [ n * i for i in range(1, 11) ]
    for num in times_table:
        print(num)
```

What is the order of growth for a call to `foo(n)`? $\Theta(1)$

(b) (1 pt) Now consider the following function definition:

```
def bar(n):
    if n == 3:
        return 'three!'
    for i in range(n // 2):
        bar(3)
```

What is the order of growth for a call to `bar(n)`? $\Theta(n)$

(c) (1 pt) Now consier the following function definition:

```
def spam(n):
    for i in range(n):
        for j in range(i):
            return spam(n - 1)
```

What is the order of growth for a call to `spam(n)`? $\Theta(n)$

9. (2 points) Newton's Method Show how you would use Newton's method to find the golden ratio ϕ. The golden ratio is defined as the positive solution to

$$\phi^2 = \phi + 1$$

Here are the functions available to you, as defined in lecture:

```
find_zero(f, df, x=1)   # Finds the zero of the function f.
deriv(f)                # Returns a function that computes f'(x)
easy_find_zero(f, x=1)  # Finds the zero of the function f.

easy_find_zero(lambda x: x*x - x - 1)
```

10. (3 points) (Extra Credit) Halting Problem

(a) (1 pt) Describe the domain and range of `will_halt` and also what `will_halt` does.

Domain: Function and arguments to that function.

Range: Boolean (True or False)

`will_halt` returns `False` if calling the function on the provided arguments would cause an infinite loop, and `True` otherwise.

(b) (2 pt) Consider the function `will_return_number`. It takes as input a function `f` and an input `x` to that function. It returns `True` if `f(x)` would evaluate to a number, and `False` otherwise. Note in particular that even if `f(x)` would cause an error or an infinite loop, `will_return_number` would still return `False`. We will use the idea that `will_halt` does not exist to prove that `will_return_number` does not exist. Fill in the blanks in the proof below:

Assume for contradiction `will_return_number` exists.

Then we can construct `will_halts` as follows:

```
def will_halt(f, x):
  def g(y):
    f(y)
    return 8
  return will_return_number(g, x)
```

But we know that `will_halt` does not exist.

So, `will_return_number` cannot exist.