CS 61A Structure and Interpretation of Computer Programs

Spring 2017

Midterm 1 Review

Introduction. I don't see any point in writing full questions for these topics – either you've already done all of the previous exam / discussion / **online** questions (and you probably don't need much more practice) or you haven't, in which case you should probably get on that. This worksheet has instead been designed to clear up misunderstandings for relevant topics, and therefore revolves around small, painless questions and descriptions (edit: ...until I got bored partway through and started writing speed overviews instead). It is neither a substitute for grinding out past exams on your own, nor one for learning this material in the first place.

I'm also obligated to mention that the topics described here aren't necessarily the exact topics you'll see on your exam. In truth you're responsible for the "sum total of human knowledge since the beginning of recorded history with particular emphasis on the contents of this course", in practice you're responsible for any material from lectures before 2/17, and in all likelihood you're responsible for lectures through 2/10 or so. But who knows?

Best of luck! Even if you shouldn't really need luck here. Sleep well, eat well, prepare your favorite writing implements, and *focus*. Be careful when running through the WWPP and environment diagram questions. Try out small examples for the programming questions (and PLEASE read the doctests!). I'll see you on the other side.

1. **(5 points) Control**

   To quote the great 14th century philosopher Peter Xu, "the concept of control revolves around truths and falsehoods." We have if statements, boolean expressions, and while loops. Of these, the trickiest things are probably the booleans since everything else reads like English.

   One important thing to remember is that a chain of `and`/`or` operators is equal to *the last expression evaluated*. Another important thing to remember is that short-circuiting is a thing; if we have a guaranteed true or false value, we stop evaluating expressions.

   (a) **(2 pt)** What would `x` in the below code evaluate to?

   ```
   x = (0 and 1 and 2) + (0 or 1 or 2)
   ```

   (b) **(2 pt)** What does this evaluate to?

   ```
   ((-4 or 0) and 4) / (-2 or (0 and 2))
   ```

   (c) **(1 pt)** How about this? Assume, as you probably would in a WWPP question, that the previous subparts have been executed in the same Python session.

   ```
   if x <= 1:
       print('hello')
   elif x <= 2:
       print(' world')
   if x <= 3:
       print(' my name is inigo montoya')
   else:
       print(' from the other side')
   ```

   ---

2. (5 points) Higher-Order Functions / Lambdas

"Higher-order function" means that at least one of two things is involved: a function being passed to another function as an argument, or a function being returned from another function. The main confusions here seem to stem from how lookup will occur within higher-order functions; for this, just remember that lookup for a name always begins in the current frame and progresses through parent frames [and parent frames correspond to the frames in which functions were *defined* (not called!)].

Also, assignment always happens in the current frame (without the `nonlocal` or `global` keyword, at least). It doesn't matter if the name has already been defined in a parent frame.

A lambda is a function value in and of itself. It does not necessarily have to be assigned to a name, as would be the case when you use `def` to define a function.

(a) (5 pt) What's the output?

```
def f(v, x):
    def g(y, z):
        return y(x, z)(z, x)
    return g
u = lambda y, x: y * 4
v = lambda x, y: x * 3 + y
f(u, 1)(lambda x, y: lambda y, x: y * 3 + v(x, y), 2)


---------------
```

(Sorry.)

3. (5 points) Recursion

Hello recursion, my old friend...

There are three types of recursion you should be familiar with: *mutual*, *tree*, and *regular old unlabeled* recursion. Mutual recursion involves two functions calling each other. Tree recursion involves multiple recursive calls in the function body. "Regular old" recursion involves a single recursive call. Every recursive definition must have some type of stopping condition – a *base case*. And almost always there will be some other processing inside the function body, in which necessary problem-specific work is done.

You can bet on having a recursive programming question on the exam. So that's what I'll talk about here.

When you're writing a recursive program, you need to remember that there'll be a recursive case and a base case... and figure out how these tie into everything. Often, this involves thinking of the base case (the *simplest* case) first – *what do you do in the simplest case?* – and then working toward it through your recursive calls (*how can you break your main problem into subproblems with simpler inputs?*).

```
def <generic_recursive_function>(<parameters>):
    if <condition that signifies a base case>:
        <handle base case>
        <return something that solves the base case>
    else:
        <do stuff>
        <make a recursive call with simpler args>
        <return something that solves the problem for the current input>
```

Of course, there might be more than one base case – or more than one situation in which you make a recursive call. If there are multiple recursive calls (as in tree recursion), it's more likely that you'll have multiple base cases.

On a test, it might help to think of a functional algorithm (i.e. how would you want to solve this problem, not necessarily in Python?) without code at first. Later, if the skeleton doesn't make any sense to you, think about how you'd write it yourself and maybe adapt your solution to the template. Otherwise, it's all about figuring out what you need to do and then doing it. Don't lose sight of the big picture for each part of your code.

4. (5 points) Environment Diagrams

Environment diagrams are just graphical representations of the "state" of a program, i.e. all of the name/value bindings. On any given line, you'll have a name and a little box. You put the value associated with the name in the little box. You make sure you put the *right* value in the little box. That's it.

How do you make sure you put the right values in the boxes? You study all the rules for program execution, all of which are outlined in the **textbook** (which I really like, incidentally) and in the many guides published on the topic.

If you go through these methodically, at a measured pace, and you know the rules (there aren't even that many), then you should be able to ace the environment diagram(s) every single time. Except maybe if the program is 100 lines long and we name every single variable `f`, but that's kind of a different problem.

Anyway, here are a couple of common mistakes. It will pain me to no end if I see them happen on your tests:

(a) Assignment (as in `x = y`) means "evaluate the expression on the right, make a copy of it, and bind the copy to the name on the left." If the "something on the right" is an arrow pointing to an object (a function, list, etc.), then the copy should be an arrow pointing to the same object. Copy the arrow, not the object!

(b) Corollary: `x = y` is not the same as `y = x`; there is no symmetry. This isn't mathematics, even though I just said "corollary."

(c) Never direct an arrow to a name. An arrow has to point to something in "object space" (the section on the right).

(d) The parent is always the frame in which the function originated (not necessarily the frame corresponding to the function *call*!).

(e) If there's no return statement, the function returns `None`. Don't forget to write this.

You should see an environment diagram, smile, and then switch into the mechanical partition of your mind that *is* the computer. In this context the computer makes no mistakes, makes no excuses, and is utterly uncreative. For these problems, that's all you have to be.

(a) (5 pt) Here, I have attempted to create a program that addresses one or two mistakes that people seem to make with environment diagrams. Give it a try (on a separate piece of paper, I guess)!

```
def f(x, h):
    def g(y, i):
        f = i[:]
        h = [f, lambda: g(5, h)]
        return h
    return g(4, h)

x, y = 6, 7
f = f(3, [lambda: x * y])
g = f[-1]()[0][0][0]()
```

Finally, a **typeset guide to environment diagrams**. It covers pretty much the same stuff as any other writeup you'll find, but it won't hurt you to skim through this for reference.

5. (5 points) Data Abstraction

Data abstraction: abstractly representing an object in Python code. The tl;dr of it is that there are several different layers – with abstraction barriers in between, of course. Nothing except the constructors and the selectors should know how the object is really represented (i.e. nothing else should assume that the object is really a list or a function or whatever). **Constructors** create and return objects. **Selectors** take in an object and return some data associated with that object. Very simple.

Two ADTs you should definitely be familiar with are the ones that we use to represent trees and linked lists. I've included these in the next two sections, sans implementations (with data abstraction, implementations shouldn't matter anyway!), if you want to refresh your memory.

6. (5 points) Trees

```
# Tree ADT

def tree(label, branches=[]):
    """Constructs a tree with label LABEL and branches BRANCHES."""

def label(tree):
    """Returns the element at the root node of the given tree."""

def branches(tree):
    """Returns the branches/subtrees of TREE in a list."""

def is_leaf(tree):
    """Returns true iff the given tree is a leaf (i.e. it has no branches)."""
```

We'll usually want to process trees recursively, since we don't know the structure of our trees ahead of time and they – being recursively-defined data structures – translate naturally to recursive programs anyway. Often, recursive tree programs will follow a pattern similar to the following:

```
def <tree_fn>(<parameters, probably incl. a tree>):
    if <base case, commonly is_leaf>:
        <return something that solves the base case>
    else:
        <make a recursive call for each of the branches>
        <do necessary work for the current tree, using the label
                and the results from each of the recursive calls>
        <return something that solves the problem for the current input>
```

7. (5 points) Linked Lists

```
# Linked list ADT

empty = 'empty'

def link(first, rest=empty):
    """Constructs a linked list from its first element and the rest."""

def first(s):
    """Returns the first element of a linked list S."""

def rest(s):
    """Returns the rest of the elements of a linked list S."""

def isempty(s):
    """Returns true iff S is the empty list."""

def print_link(s):
    """Prints elements of a linked list S.

    >>> s = link(1, link(2, link(3, empty)))
    >>> print_link(s)
    1 2 3
    """
```

Linked lists are, like trees, recursive data structures. This is because the `rest` of a linked list is always another linked list (if we consider `empty` to be a linked list as well). Due to this, they lend well to recursion. Since linked lists are generally linear (unless we start storing linked lists as `first` elements), they can usually be processed fairly easily with iteration as well.

A typical recursive pattern for a linked list program might be

```
def <ll_fn>(<parameters, probably incl. a linked list>):
    if <base case, commonly isempty and/or isempty(rest(lnk))>:
        <return something that solves the base case>
    else:
        <make recursive call(s) for some combination of "rest" linked lists>
        <process the linked list's first element in some way
                [maybe combine this step with the recursive call(s)]>
        <return something that solves the problem for the current input>
```

Unfortunately, because these patterns are so commonplace they are less likely to appear on an exam. At least not without some twist...

8. (5 points) Miscellaneous

Returning gives a value back to the program. Printing displays a value for the user.
Returning breaks out of a function call. Printing doesn't.
`print(x, y)` will print `x` and `y` separated by a space.
Returned values are only displayed for a top-level function call typed into the interpreter.
Printed values are pretty much always displayed.
The interpreter won't show anything if you return `None`. (It will if you *print* `None`.)

Function calls are denoted by parentheses `(...)` after a function name.
Parentheses around a value [à la `(3) + (((4)))`, which equals `7`] are just being used for grouping.
...although if there are commas involved, it's a tuple [à la `(3, 4) + (((5,)))`, which equals `(3, 4, 5)`].

Last thing: this might be obvious, but don't leave things blank if you're not sure... we do give partial credit! In the worst case, just write down something random; this can only hurt you if you have too much pride. (Such would be a misconception. In 61A, there is no pride.[1])

Have a good time with it!

- oj

[1] That doesn't mean copy off of your neighbor's paper, though. Your neighbor is definitely wrong.