Problem Solving with the Structure and Interpretation of Computer Programs

by Winston Purnomo, University of California-Berkeley

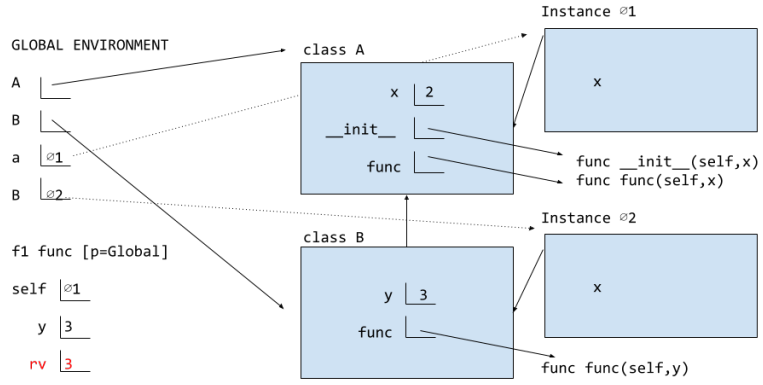
December 19, 2019

Contents

1 Object-oriented Programming

61A never formally teaches you the best way to do object-oriented programming environment diagrams. It means that it's up to you to figure out the best way to do this, and the methods I lay out here are the ones that I personally find most useful:

```
class A:
    x = 2

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

    def func(self,y):
        return y

class B(A):
    y = 3

    def func(self,y):
        return y+1

a = A(3)
b = B(1)
a.func(3)
```



We continue to use the standard environment diagram model. Classes accessible from the global environment are placed in the frame like above.

- Classes that inherit from another class have an arrow denoting which class they inherit from.
- Instance and class variables are sufficiently separated: we draw an arrow from instances to the right side of the class they are from.
- Each instance is given a unique identifier like a frame in a function. Unfortunately since O1 (Object 1) and I1 (Instance 1) look far too much like 01 and 11, we didn't use those prefixes. What I've settled on using is a crossed-out O, or the mathematical representation of an empty set, because I first started writing O1, got confused, and just added slashes to the middle. You can either draw the arrow from the name an instance is bound to, or if your diagram is too cluttered, just write the name of the instance as I did. The arrows are dotted to denote it is optional.
- Function calls are still opened on the left side of the diagram, below the environment diagrams. self is always a denoted variable even when it is a bound method, and we use the unique identifier we came up with before to identify these instances we are referring to.
- Doing this, we can always trace values from the global environment, to the instance or class it is referring to, to their parent class or superclass, etc. It takes much of the guesswork out of OOP WWPD questions.

2 Recursion

Recursion is a method of problem-solving in which a function calls on itself within its body.

Tree recursion is a method in which a function calls on itself twice or more within its body.

2.1 Permutation type questions

Any problem where we are dealing with a permutation, it is usually simplest done through recursion, or tree recursion.

Here is the general structure of the permutation type question:

```
def function(x):
    if <base case>:
        ...
    else:
        a = <permutation 1>
        b = <permutation 2>
        # There is usually a list comprehension somewhere to
          permutate.
        return a + b
```

Usually, permutation 1 advances the recursive case forward by 1, and permutation 2 advances the recursive case forward by 2.

2.1.1 Question 1 (Fall 2019, Practice Midterm 2)

Write a function no_eleven that returns a list of all distinct length-n lists of ones and sixes in which 1 and 1 do not appear consecutively.

Solution

```
def no_eleven(n):
    if n == 0:
        return [[]]
    elif n == 1:
        return [[6], [1]]
    else:
        a, b = no_eleven(n-2), no_eleven(n-1)
        return [[6] + s for s in a] + [[1, 6] + s for s in b]]
```

You can see how well our general structure above works in this question. We are literally combining the two permutations by using tree recursion. By only ever adding 1 when 6 is immediately after, we can be confident we will never be adding two 1s together.

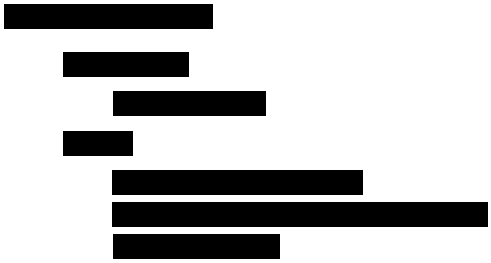
2.1.2 Question 2 (Fall 2019, Lab 8)

A subsequence of a sequence S is a sequence of elements from S, in the same order they appear in S, but possibly with elements missing. Thus, the lists [], [1, 3], [2], and [1, 2, 3] are some (but not all) of the subsequences of [1, 2, 3]. Write a function that takes a list and returns a list of lists, for which each individual list is a subsequence of the original input.

In order to accomplish this, you might want to use the function insert_into_all defined below:

```
def insert_into_all(item, nested_list):


    “"""Assuming that nested_list is a list of lists,
    return a new list consisting of all the lists in
    nested_list, but with item added to the front of
    each.
    >>> nl = [[], [1, 2], [3]]
    >>> insert_into_all(0, nl)
    [[0], [0, 1, 2], [0, 3]]
    """
    return [[item] + lst for lst in nested_list]
```

Solution



Again, our defined structure makes it work! We could have not even used insert_into_all to design this solution because it is fundamentally very similar to the previous question.

By using this general structure, we should always be able to answer this kind of question in tests.

2.2 Tracker variables

In certain recursive functions, we use tracker variables by calling the function or a helper function multiple times:

Here is the general structure for this type of question without a helper function:

```
def function(data, k):
    if k == 0:
        return 0
    elif data == ...:
        return function(data, k-1)
    else:
        return function(data,k)
```

And here is without:

```
def function(data):
    def helper(data, value):
        if value == 0:
            return ...
        else:
            if data == ...:
              return helper(data, value-1)
            else:
              return helper(data, value)
```

```
    return helper(data, k) #where k is some value we want to track
```

Note that this is not considered tree recursion; while there are two function calls of the helper functions within its own body, it is never the case that both calls are executed. (Not important to memorize, but good to know regardless)

This type of structure essentially posits a base case, and an if situation for the recursive case: if the conditional passes, then the function is advanced and the tracker variable is updated to reflect this, and if it fails, then the function is advanced without updating the tracker variable.

2.2.1 Question 1 (Fall 2019, CSM Week 9, Modified)

Given a list of values, write a function contains that returns True only if there are exactly n instances of elem in the list:

Solution

```
def contains(elem,n,lst):
    if n == 0:
        return True
    elif n == 1 and lst[0] == elem and elem not in lst[1:]:
        return True
    elif lst[0] == elem:
        return contains(elem,n-1,lst[1:])
    else:
        return contains(elem,n,lst[1:])
```

2.2.2 Question 2 (Fall 2019, Lab 8)

This question makes use of both tracker variables and permutations.

In Lab 4, we examined the Subsequences problem. A subsequence of a sequence S is a sequence of elements from S, in the same order they appear in S, but possibly with elements missing. For example, the lists [], [1, 3], [2], and [1, 3, 2] are subsequences of [1, 3, 2]. Again, we want to write a function that takes a list and returns a list of lists, where each individual list is a subsequence of the original input.

This time we have another condition: we only want the subsequences for which consecutive elements are nondecreasing. For example, [1, 3, 2] is a subsequence of [1, 3, 2, 4], but since $2 < 3$, this subsequence would not be included in our result.

Fill in the blanks to complete the implementation of the inc_subseqs function. You may assume that the input list contains no negative elements.

You may use the provided helper function insert_into_all, which takes in an item and a list of lists and inserts the item to the front of each list.

Solution

This is tricky, because it is a tree recursion problem that calls itself **three times**, but again, we are fundamentally doing the same thing.

We define a base case, then establish a special case for which the problem has defined. If the number in the sequence is smaller than the previous number we are tracking, then we will skip to the next number in the sequence.

What about the case that the previous number was too big? Well, we are keeping that possibility in the recursive cases below. a advances the recursive case by moving forward in the list and marking s[0] as the previous value (using insert_into_all to insert s[0] into the permutations of that return value), while b advances the recursive case by moving forward two elements.

3 Trees

A tree is a widely-used abstract data type. There are two ways to make a tree:

- As an abstract data type
- As a class

Here is the ADT definition of a tree:

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label]+branches
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```

```
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
def is_leaf(tree):
    return not branches(tree)
def leaves(t):
    if is_leaf(t):
        return [label(t)]
    else:

        return sum([leaves(b) for b in branches(t)],[])
```

And here is the class definition of a Tree:

```
class Tree:
    def __init__(self,label,branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch,Tree)
        self.branches = list(branches)
def is_leaf(self):
    return not self.branches
def leaves(t):
    if t.is_leaf():
        return [t.label]
    else:return sum([leaves(b) for b in t.branches],[])
```

The ADT definition of the function is_tree is a common example of a recursive solution to a tree question:

3.1 The For-Until Solution

The for-until solution does not have an explicit base case. It relies on an implicit base case. It uses a for/while loop and an if statement containing a recursive call within its suite, during which point it will return a False statement if any of the conditions are hit. If the code manages to make it through the entire loop without hitting a return False statement, it will return True at the end.

General structure:

```
def function(tree):
    ...
    for/while <...>:
        if <...>:
            return False
    return True
```

This structure is only generally used when you want to return a boolean value.

3.1.1 Question 1 (Guerilla 2, Fall 2019)

A min-heap is a tree with the special property that every nodes value is less thanor equal to the values of all of its children. For example, the following tree is a min-heap:

However, the following tree is not a min-heap because the node with value 3 has avalue greater than one of its children:

Write a function is_min_heap that takes a tree and returns True if the tree is a min-heap and False otherwise.

Solution (uses ADT definition):

```
def is_min_heap(t):
    for b in branches(t):

        if label(t) > label(b) or not is_min_heap(b):
            return False
    return True
```

3.2 The Basic Recursion Solution

A very general structure that is primarily used in Tree solutions is:

```
def function(t):
    if is_leaf(t):
        return ...
    else: #The else clause is optional.
        for ... in branches(t):

            ... # A recursive call to function is usually found here
            return ... # or here
```

Occassionally, you may also meet this general structure:

```
def function(t):
    if is_leaf(t):
        return ...
    else:

        placeholder = [function(b) for b in branches(t)]
        return label(t)... # Usually either the placeholder is
                             combined with or compared against
                             t.label.
```

At a fundamental level, you will see below that the solutions all share many similarities, even if they have been written differently!

3.2.1 Question 1 (Midterm 2, Fall 2015)

Implement complete, which takes a Tree instance t and two positive integers, d and k. It returns whether t is d-k-complete. A tree is d-k-complete if every node at a depth less than d has exactly k branches and every node at depth d is a leaf. Notes: The depth of a node is the number of steps from the root; the root node has depth 0. The built-in all function takes a sequence and returns whether all elements are true values: all([1, 2]) is True but all([0, 1]) is False. Tree appears on the Midterm 2 Study Guide.

Solution (uses Class definition)

```
def complete(t, d, k):
    if not t.branches: #Same as t.is_leaf()
        return d == 0
    bs = [complete(b, d-1,k) for b in t.branches]
    return len(bs) == k and all(bs)
```

3.2.2 Question 2 (Midterm 2, Spring 2016)

Referring back to the explanation of min-heap from the For-Until loop question, write a function remove_leaf to destructively remove the far leftmost leaf of a tree and return its label.

Solution (uses Class definition)

```
def remove_leaf(t):
    child = t.branches[0]
    if child.is_leaf():
        v = child.label
        t.branches = t.branches[1:]
        return v
    else:
        return remove_leaf(child)
```

3.2.3 Question 3 (Midterm 2, Spring 2018)

A sibling of a node in a tree is another node with the same parent. Implement sibling, which takes a Tree instance t, and returns a lsit of the labels of nodes in t that have a sibling. For example

```
          1
         / \
        3   9
        |   |
        4   5
       / | \
      5  3  8
Result = [3,9,5,3,8]
```

Solution (uses class definition)

```
def siblings(t):

    result = [b.label for b in t.branches if len(t.branches) > 1]
    for b in t.branches:
        result += siblings(b)
    return result
```

3.3 The Zip Solution

We have never seen this in any past iteration of 61A, but I'm willing to bet that a question with a zip function will be implemented.

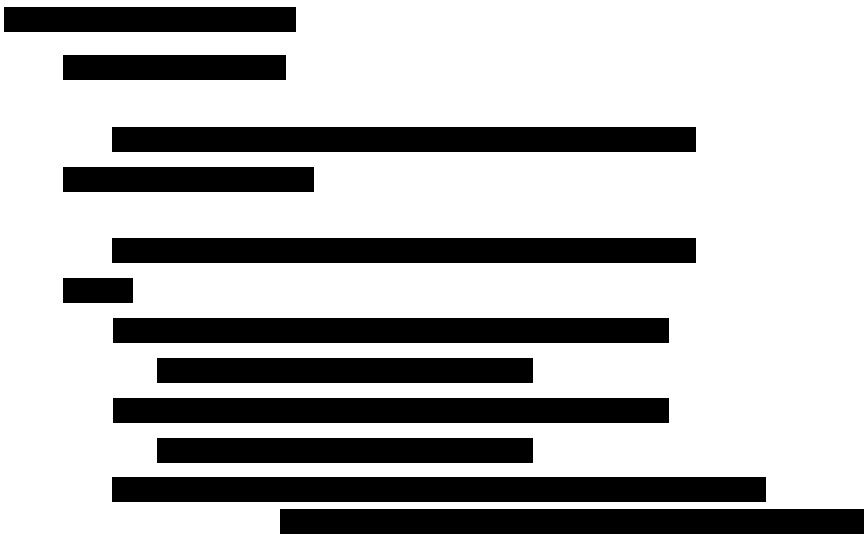
The zip function takes two lists and returns an iterator of tuples where the first item in each passed-in iterator is paired together.

```
a = [1,2,3]
b = [4,5,6]
x = zip(a,b)
print(list(x))
[[1,4],[2,5],[3,6]]
```

3.3.1 Question 1 (Fall 2019, Lab 5, Modified)

Implement add_trees, which takes two trees and returns a new tree where each corresponding element in the two trees are added together. If one tree has elements in a place where the other element does not, that element should be returned without being modified.

Solution (uses class definition)



The zip function takes the branches of both t1 and t2 and lets the for loop iterate over both elements.

4 Linked Lists

A linked list is an inherently recursive data structure. In 61A, they are always represented with the Link class:

```
class Link:

    def __init__(self,first,rest=empty):
        empty = ()
        self.first = first
        self.rest = Link(rest)
```

4.1 The Recursive Solution

The recursive solution is the most common type of linked list solution.

```
def function(lst):
    if lst is Link.empty:
        ...
    else:
        ...
        function(lst.rest)
```

Some solutions use lst.rest == Link.empty, which depends on whether you will need to do anything to the final label before the empty list.

This basic structure solves more than half of linked list questions with some modification.

4.1.1 Question 1 (Spring 2015, Midterm 2)

Implement double_up, which mutates a linked list by inserting elements so that each element is adjacent to an equal element. The double_up function inserts as few elements as possible and returns the number of insertions.

Solution

```
def double_up(s):
    if s == Link.empty:
        return 0
    elif s.rest == Link.empty:
        s.rest = Link(s.first,s.rest)
        return 1
    elif s.first == s.rest:
        return double_up(s.rest.rest)
    else
        s.rest = Link(s.first,Link.empty)
        return 1 + double_up(s.rest.rest)
```

You can see that although this involves an if and two elif clauses, the basic structure of the solution remains the same. There is one suite of conditions that handles what happens when an empty list is returned (which means we've hit the end), and another suite that handles what to do before we get to the end.

4.2 The Iterative Solution

It is possible to perform the iterative solution on linked lists, which follows this general structure:

```
def function(lst):
    ...
    while lst.rest: #lst and lst != Link.empty are also
                    both possible
        ...
        lst = lst.rest
    return ...
```

The key difference with the recursive definition is that the iterative solution moves through the list front to back, while the recursive definition moves from back to front.

4.2.1 Question 1 and 2 (Spring 2016, Midterm 2)

Similarly to double-up above, implement the functions double1 and double2, which produce linked lists in which each item of the original list is repeated immediately after that item:

- double1 should be non-destructive, producing a new list without disturbing the old list.
- double2 should be destructive, modifying the original linked list wherever possible.

Solution

```
def double1(s):
    result = Link.empty
    last = None
    while s is not Link.empty:
        if last is None:

            result = Link(L.first,Linked(L.first))
            last = result.rest
        else:

            last.rest = Link(L.first, Link(L.first))
            last = last.rest.rest
        s = s.rest
    return result
```

```
def double2(s):
    result = s
    while s is not Link.empty:
        s.rest = Link(s.first,s.rest))
        s = s.rest.rest
    return result
```

4.2.2 Question 3 (Spring 2019, Midterm 2)

Implement link_to_dict which takes a linked list encoding a "flattened" dictionary (in which the elements are key 1 > value 1 > key 2 > value 2 etc.), removes all the values, and returns the equivalent dictionary. The input and returned list may include duplicate keys. You may assume the linked list always has an even number of elements:

```
def link_to_dict(s):
    d = {}
    while L is not Link.empty:
        key, value = L.first, L.rest.first
        if key not in D:
            D[key] = [value]
        else:
            D[key].append(value)
        L.rest, L = L.rest.rest, L.rest.rest
```

5 Miscellaneous knowledge

5.1 Decorators

Here is how the memo-ization function works:

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

You can memoize any function with @memo, or by passing it into the higher-order function memo:

```
>>> fib = memo(fib)
>>> fib(2)
1
```

Every decorator you need to build in this class will follow this same general structure:

```
def decorator(function):
    def decorated(value):
        ...
        return function(value)
    return decorated
```

6 Macros

Macros are a special form in Scheme that mix up the order of operations. Here is how Scheme typically executes procedures:

1. Evaluate the operator.
2. Evaluate all of the operands, left to right.
3. Apply the operator to the operands by evaluating the body of the procedure.

Macro procedures are executed as such:

1. Evaluate the operator.
2. Apply the operator to the unevaluated operands, creating a new Scheme program that is "returned" by the procedure.
3. Execute said program within the frame it was called in.

6.0.1 Question 1 (Spring 2018, Final)

Implement lambda-macro, a macro that creates anonymous macros. Alambda-macroexpression hasa list of formal parameters and one body expression. It creates a macro with those formal parameters andthat body. Assume that the symbol anon is not use anywhere else in a program that contains lambda-macro.

```
(define-macro (lambda-macro bindings body)
    '(begin (___ ___ ___) anon))
```

Solution

```
(define-macro (lambda-macro bindings body)
```

```
'(begin (define-macro ,(cons 'anon bindings) ,body) anon))
```

Why does this work? Let's see what the unevaluated Scheme procedure is for a program ((lambda-macro (x) (car x)) (+ 1 2)):

```
(begin (define-macro (anon x) (car x)) anon)
```

You will see that first we define this new macro function and give it the name anon. Then, the anon name is returned into the global frame, which means we can directly use it (in the sample program above, it means we are calling:

```
(anon (+ 1 2))
```

Why do we use cons? Well, remember that bindings in lambda functions are passed in as a list of their own, while in a def, they are passed in as a list **together with** the name of the procedure:

```
(define (x a b) ...)
(lambda (a b) ...)
```

The cons allows us to easily "merge" the name anon with this list of parameters.

6.0.2 Question 2 (Summer 2018, Mock Final)

Write a macro called zero-cond that takes in a list of clauses, where each clause is a two-element list containingtwo expressions, a predicate and a corresponding result expression. All predicates evaluate to a number. The macro should evaluate each predicate and return the value of the expression corresponding to the first true predicate, treating 0 as a false value.

```
scm> (zero-cond((0 'result1)
...  ((- 1 1)'result2)
...  ((* 1 1)'result3)
...  (2 'result4)))
result3
(define-macro (zero-cond clauses)
(cons 'cond (map _________________________________
_________________________________________________
)))
```

Solution

```
(define-macro (zero-cond clauses)
    (cons 'cond
        (map
            (lambda (clause) (cons
```

```
            (not (= 0 (eval (car clause) ) ) )
            (cdr clause) )
          )
          clauses)))
```

Why does this work? First, let's see what the unevaluated Scheme program is (before step 3 is executed) for the sample code above.

```
(cond
    (#f 'result1)
    (#f 'result2)
    (#t 'result3)
    (#f 'result4)
)
```

This question is very unique because it involves both a map and an inner lambda function in its solution. The lambda function requires an inner eval because it is taking in the clause variable as an unevaluated operator from the macro, and quoting and unquoting... doesn't really go that far.

As for the other parts, remember that map takes in a function (our lambda) and an iterable (clauses), and returns a new list, the cond list we build. Our lambda takes each of the clauses, evaluates its car (the predicate), and then returns a list that represents a single clause in cond, where the consequent is not evaluated at this stage. We use cons in the lambda because we want (cdr clause) to be part of the same list as the predicate, and using cons in the beginning because our map already returns a beautiful list, and we just need to merge it with the single cond special form at the beginning.

6.0.3 Question 3 (Spring 2019, Final)

The if special form has been removed from Scheme. Implement an if macro using only and/or.

```
(define-macro (if _______) ________)
```

Solution

```
(define-macro (if pred conq alt) `(or (and ,pred ,conq)
(and (not ,pred) ,alt)))
```

This is probably the most straightforward macro question of all, because it frankly isn't testing your macro knowledge so much as testing your logic. Nevertheless, it is a good study in why we should use macros, as it is a program that would not otherwise be possible without it.

7 Streams