In [ ]:
```
#utility to help visualize recursive calls (on stderr)
from instrument import instrument
```

In [ ]:
```
instrument.SHOW_CALL = True
instrument.SHOW_RET = True
```

Recursion and Recursive Patterns

What is recursion?

Generally, recursion occurs when something is defined in terms of itself.

Consider `factorial` in a mathematical sense:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

We can implement that idea recursively in python:

In [ ]:
```
def factorial(n):
    if n == 0:                #base case
        return 1
    return n * factorial(n-1) #recursive case
```

In [ ]:
```
```

Ways to visualize recursive evaluation

Environment Model: With the environment model, we can emulate what happens for something like `factorial(2)` -- and we see something very important and interesting. We see that the python *stack* keeps track of intermediate (partial) results while the recursive call does its work, and then when that recursive call returns, python finishes up by combining that returned result with the intermediate (partial) result.

Call/return tree: An abstracted visualization/sketch of call (perhaps with intermediate calculations), and recursive calls as children of current function call. Recursion depth corresponds to depth of the tree.

Function entry and exit (our @instrument decorator): A printed version of call entries and exits, with recursion depth shown by indentation.

In [ ]:
```
@instrument
def factorial(n):
    if n == 0:                #base case
        return 1
    return n * factorial(n-1) #recursive case
```

In [ ]:
```
factorial(2)
```

Recursive patterns often include those operating on data structures (not just numbers).

Walk the list to find the first value satisfying function `f`

In [ ]:
```
@instrument
def walk_list(L, f):
    """Walk a list -- in a recursive style. Note that this is done as a
    stepping stone toward other recursive functions, and so does not
    use easier/direct built-in list functions.

    In this first version -- walk the list just to find/return the
    FIRST item that satisfies some condition, where f(item) is true.

    >>> walk_list([1, 2, 3], lambda x: x > 2)
    3
    """
    pass
```

In [ ]:
```
walk_list([1, 2, 3], lambda x: x > 2)
```

Walk a list, but now returning a *list* of items that satisfy `f` -- uses stack

Here the stack is used to remember and build on intermediate results.

In [ ]:
```
@instrument
def walk_list_filter1(L, f):
    """ Walk a list, returning a list of items that satisfy the
    condition f.

    This implementation uses the stack to hold intermediate results,
    and completes construction of the return list upon return of
    the recursive call.

    >>> walk_list_filter1([1, 2, 3], lambda x: x % 2 == 1) #odd only
    [1, 3]
    """
    pass
```

In [ ]:
```
walk_list_filter1([1, 2, 3], lambda x: x % 2 == 1)
```

Walk a list, returning a list of items that satisfy `f` -- uses helper with a "so_far" argument

In [ ]:
```python
@instrument
def walk_list_filter2(L, f):
    """ Walk a list, returning a list of items that satisfy the
    condition f.

    This implementation uses a helper with an explicit 'so far'
    variable, that holds the return value as it is being built
    up incrementally on each call.

    >>> walk_list_filter2([1, 2, 3], lambda x: x % 2 == 1)
    [1, 3]
    """
    @instrument
    def helper(L, so_far):
        pass

    return helper(L, [])
```

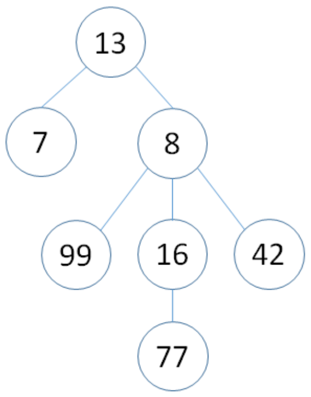
In [ ]:
```python
walk_list_filter2([1, 2, 3], lambda x: x % 2 == 1)
```

Note the difference in how this works. `walk_list_filter2` builds up the result as an evolving argument to `helper`. When we're done, the stack does nothing more than keep passing that result back up the call chain (i.e., is written in a tail-recursive fashion). In contrast, `walk_list_filter1` uses the stack to hold partial results, and then does further work to build or complete the result after each recursive call returns.

Now consider some functions that recurse on trees...

We want to extend the basic idea of recursive walkers and builders for lists, now to trees. We'll see the same patterns at work, but now often with more base cases and/or more recursive branch cases.

For these examples, we need a simple tree structure. Here we'll represent a node in a tree as a list with the first element being the node value, and the rest of the list being the children nodes. That is to say, our tree structure is a simple nested list structure.



In [ ]:
```
tree1 = [13,
         [7],
         [8,
          [99],
          [16,
           [77]],
          [42]]]
tree1
```

In [ ]:
```
@instrument
def tree_max(tree):
    """Walk a tree, returning the maximum value in the
       (assumed non-empty) tree.
    >>> tree_max([13, [7], [8, [99], [16, [77]], [42]]])
    99
    """
    pass
```

In [ ]:
```
tree_max(tree1)
```

In [ ]:
```
@instrument
def depth_tree(tree):
    """ Walk a tree, returning the depth of the tree
    >>> depth_tree([13, [7], [8, [99], [16, [77]], [42]]])
    4
    """
    pass
```

In [ ]:
```
depth_tree([13, [7], [8, [99], [16, [77]], [42]]])
```

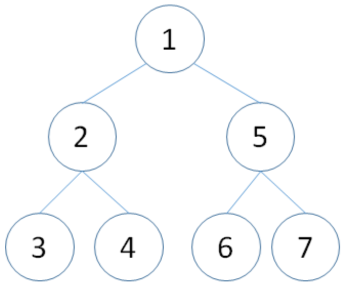
In [ ]:
```
depth_tree._max_depth #@instrument tells us max recursion depth
```

Notice that the recursion structure is exactly the same in both cases? We could generalize to something like a `walk_tree` that took a tree *and* a function `f` (and perhaps some other base case values), and did that operation at each step. We'll leave that as an exercise for the reader.

Now a "builder" or "maker" function, that recursively *creates* a tree structure...

This will be a simple binary tree with left and right branches balanced in terms of the number of nodes in each branch.

`make_tree([1,2,3,4,5,6,7])` should return a tree `[1, [2, [3], [4]], [5, [6], [7]]]` corresponding to:



In [ ]:
```python
@instrument
def make_tree(L):
    """ Make and return a binary tree corresponding to the list. The
    tree is "binary" in the sense that the number of left and right
    branches are balanced as much as possible, but no condition is
    imposed on the left/right values under each node in the tree.

    >>> make_tree([1,2,3,4,5,6,7])
    [1, [2, [3], [4]], [5, [6], [7]]]
    >>> make_tree([1,2])
    [1, [2]]
    """
    n = len(L)
    pass
```

In [ ]:
```python
tree2 = make_tree([1, 2, 3, 4, 5, 6, 7])
tree2
```

In [ ]:
```python
tree3 = make_tree([1, 2]) #unbalanced tree case
tree3
```

How many calls to `make_tree` do you expect for a list of length `n`?

In [ ]:
```python
instrument.SHOW_CALL = True
instrument.SHOW_RET = False
```

In [ ]:
```python
make_tree._count = 0 #enabled by @instrument
make_tree._max_depth = 0

tree4 = make_tree(list(range(8)))
tree4
```

In [ ]:
```python
make_tree._count
```

In [ ]:
```python
make_tree._max_depth
```

Another recursive example -- a printed visualization of a tree

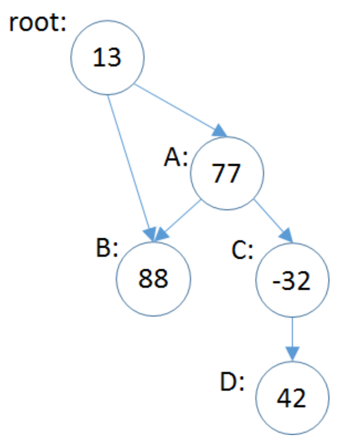
In [ ]:
```
def show_tree(tree):
    """ Return a formatted string representation to visualize a tree """
    spaces = '    '
    @instrument
    def helper(tree, level):
        if not tree:
            return ""
        val = tree[0]
        children = tree[1:]
        result = spaces*level + str(val) + '\n'
        for child in children:
            result += helper(child, level+1)
        return result
    return helper(tree, 0)
```

In [ ]:
```
print("tree4:", tree4, "\n", show_tree(tree4))
```

This `show_tree` implementation is actually very similar to the recursive structure used inside our `@instrument` decorator! Feel free to look at that code and to use `instrument.py` in your own debugging, if you'd like.

Finally, consider some functions that recurse on directed graphs...

For this, we need a more sophisticated structure, since a node may be referenced from more than one other node. We'll represent a directed graph (also known as a "digraph") as a dictionary with node names as keys, and associated with the key is a list holding the node value and a list of children node names. The special name 'root' is the root of the graph.



In [ ]:
```
graph1 = {'root': [13, ['A', 'B']],
          'A': [77, ['B', 'C']],
          'B': [88, []],
          'C': [-32, ['D']],
          'D': [42, []]}
```

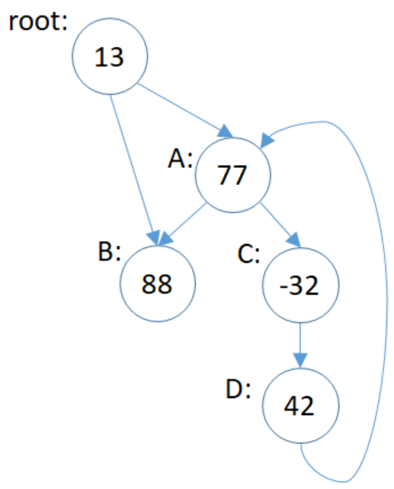
In [ ]:
```
@instrument
def graph_max(graph):
    """Walk a graph, returning the maximum value in a (non-empty) graph.
    First, we'll assume there are no cycles in the graph.
    """
    @instrument
    def node_max(node_name):
        pass

    return node_max('root')
```

In [ ]:
```
instrument.SHOW_CALL = True
instrument.SHOW_RET = True
```

In [ ]:
```
graph_max(graph1)
```

What do we do if there *are* cycles in the graph? E.g.,



In [ ]:
```
graph2 = {'root': [13, ['A', 'B']],
          'A': [77, ['B', 'C']],
          'B': [88, []],
          'C': [-32, ['D']],
          'D': [42, ['A']]} #changed; now D -> A
```

In [ ]:
```
#graph_max(graph2)

## breaks (infinite recursion)!
## (need to re-execute def graph_max afterwards for instrumentation)
```

```
In [ ]: @instrument
        def graph_max2(graph):
            """Walk a graph, returning the maximum value in a (non-empty) graph.
            Now, however, there might be cycles, so need to be careful not to
            get stuck in them!
            """
            visited = set()
            @instrument
            def node_max(node_name):
                visited.add(node_name)
                val = graph[node_name][0]
                children = graph[node_name][1]
                new_children = [c for c in children if c not in visited]
                if new_children:
                    return max(val, max(node_max(child) for child in new_children))
                return val
            return node_max('root')
```

```
In [ ]: graph_max2(graph2)
```

Circular Lists

It's possible to create a simple python list that has itself as an element. In essence, that means that python lists themselves might be "graphs" and have cycles in them, not just have a tree-like structure!

```
In [ ]: x = [0, 1, 2]
        x[1] = x
        print("x:", x)
        print("x[1][1][1][1][1][1][1][1][1][1][2]:", x[1][1][1][1][1][1][1][1][1][1][2])
```

We'd like a version of `deep_copy_list` that could create a (separate standalone) copy of a recursive list, *with the same* structural sharing (including any cycles that might exist!) as in the original recursive list.

```
In [ ]: @instrument
        def deep_copy_list(old, copies=None):
            if copies is None:
                copies = {}

            oid = id(old)      #get the unique python object-id for old

            if oid in copies:  #base case: already copied object, just return it
                return copies[oid]

            if not isinstance(old, list):  #base case: not a list, remember & return it
                copies[oid] = old
                return copies[oid]

            #recursive case
            copies[oid] = []
            for e in old:
                copies[oid].append(deep_copy_list(e, copies))
            return copies[oid]
```

In [ ]:
```
y = deep_copy_list(x)
y[0] = 'zero'
print("x:", x)
print("y:", y)
print("y[1][1][1][1][1][1][1][1][1][1][2]:", y[1][1][1][1][1][1][1][1][1][1][2])
```

In [ ]:
```

```