In [1]:
```
from instrument import instrument #utility to help visualize recursive calls (on stderr)
```

In [2]:
```
instrument.SHOW_CALL = True
instrument.SHOW_RET = True
```

# Recursive Patterns

## Let's start with some simple functions that recurse on lists...

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

In [3]:
```
@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
    """
    if L == []:      #base case
        return None
    if f(L[0]):      #another base case
        return L[0]
    return walk_list(L[1:], f) #recursive case
```

In [4]:
```
walk_list([1, 2, 3], lambda x: x > 2)
```

```
call to walk_list: [1, 2, 3], <function <lambda> at 0x7f58089a3c80>
   call to walk_list: [2, 3], <function <lambda> at 0x7f58089a3c80>
      call to walk_list: [3], <function <lambda> at 0x7f58089a3c80>
      walk_list returns: 3
   walk_list returns: 3
walk_list returns: 3
```

Out[4]: 3

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

In [5]:
```python
@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]
    """
    if L == []:
        return []
    if f(L[0]):
        # the following waits to build (and then return) the list
        # until after the recursive call comes back with a sub-result
        return [L[0]] + walk_list_filter1(L[1:], f)
    else:
        return walk_list_filter1(L[1:], f)
```

In [6]:
```python
print(walk_list_filter1([1, 2, 3], lambda x: x % 2 == 1))
```

```
[1, 3]

call to walk_list_filter1: [1, 2, 3], <function <lambda> at 0x7f58080a1158>
   call to walk_list_filter1: [2, 3], <function <lambda> at 0x7f58080a1158>
      call to walk_list_filter1: [3], <function <lambda> at 0x7f58080a1158>
         call to walk_list_filter1: [], <function <lambda> at 0x7f58080a1158>
         walk_list_filter1 returns: []
      walk_list_filter1 returns: [3]
   walk_list_filter1 returns: [3]
walk_list_filter1 returns: [1, 3]
```

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

In [7]:
```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, ans_so_far):
        if L == []:
            return ans_so_far
        if f(L[0]):
            ans_so_far.append(L[0])
        return helper(L[1:], ans_so_far) #tail recursive

    return helper(L, [])
```

In [8]:
```
walk_list_filter2([1, 2, 3], lambda x: x % 2 == 1)
```

```
call to walk_list_filter2: [1, 2, 3], <function <lambda> at 0x7f58080a1950>
call to helper: [1, 2, 3], []
   call to helper: [2, 3], [1]
      call to helper: [3], [1]
         call to helper: [], [1, 3]
         helper returns: [1, 3]
      helper returns: [1, 3]
   helper returns: [1, 3]
helper returns: [1, 3]
walk_list_filter2 returns: [1, 3]
```

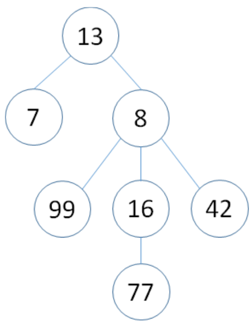
Out[8]:
```
[1, 3]
```

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 [1]:
```
tree1 = [13,
         [7],
         [8,
          [99],
          [16,
           [77]],
          [42]]]
tree1
```

Out[1]:
```
[13, [7], [8, [99], [16, [77]], [42]]]
```

In [10]:
```
@instrument
def depth_tree(tree):
    """ Walk a tree, returning the depth of the tree
    >>> depth_tree([13, [7], [8, [99], [16, [77]], [42]]])
    4
    """
    if tree == []:        #base case
        return 0

    children = tree[1:]
    if not children:      #base case
        return 1

    #recursive case
    return max(1+depth_tree(child) for child in children)
```

In [11]:
```
depth_tree([13, [7], [8, [99], [16, [77]], [42]]])
```

```
call to depth_tree: [13, [7], [8, [99], [16, [77]], [42]]]
   call to depth_tree: [7]
   depth_tree returns: 1
   call to depth_tree: [8, [99], [16, [77]], [42]]
      call to depth_tree: [99]
      depth_tree returns: 1
      call to depth_tree: [16, [77]]
         call to depth_tree: [77]
         depth_tree returns: 1
      depth_tree returns: 2
      call to depth_tree: [42]
      depth_tree returns: 1
   depth_tree returns: 3
depth_tree returns: 4
```

Out[11]: 4

In [12]:
```
@instrument
def tree_max(tree):
    """Walk a tree, returning the maximum value in the (assumed non-empty) t
ree.
    >>> tree_max([13, [7], [8, [99], [16, [77]], [42]]])
    99
    """
    val = tree[0]
    children = tree[1:]
    if not children:      #base case
        return val
    # recursive case. Note that the following launches
    # MULTIPLE recursive calls, one for each child...
    return max(val, max(tree_max(child) for child in children))
```

In [13]:
```
tree_max(tree1)
```

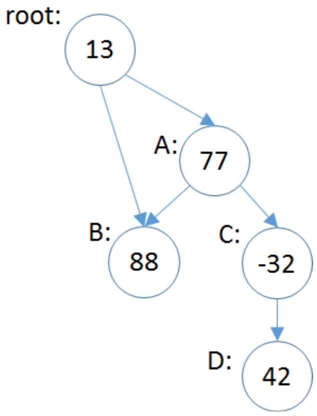
```
call to tree_max: [13, [7], [8, [99], [16, [77]], [42]]]
   call to tree_max: [7]
   tree_max returns: 7
   call to tree_max: [8, [99], [16, [77]], [42]]
      call to tree_max: [99]
      tree_max returns: 99
      call to tree_max: [16, [77]]
         call to tree_max: [77]
         tree_max returns: 77
      tree_max returns: 77
      call to tree_max: [42]
      tree_max returns: 42
   tree_max returns: 99
tree_max returns: 99
```

Out[13]: 99

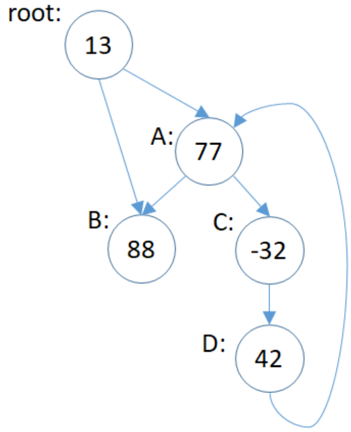
Notice that the recursion structure is exactly the same in both cases? We could generalize to something like a `walk_tree` that took a tree *and* a function `f` (and perhaps some other base case values), and did that operation at each step. We'll leave that as an exercise for the reader.

## 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.



Moreover, graphs may also contain cycles! E.g.,




How do we avoid infinite recursion?

In [14]:
```
graph1 = {'root': [13, ['A', 'B']],
          'A': [77, ['B', 'C']],
          'B': [88, []],
          'C': [-32, ['D']],
          'D': [42, []]}
```

In [15]:
```
graph2 = {'root': [13, ['A', 'B']],
          'A': [77, ['B', 'C']],
          'B': [88, []],
          'C': [-32, ['D']],
          'D': [42, ['A']]}  # D --> A
```

In [16]:
```
@instrument
def graph_max(graph):
    """Walk a graph, returning the maximum value in a (non-empty) graph.
    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 [17]:
```
instrument.SHOW_CALL = True
instrument.SHOW_RET = True
```

In [18]:
```
graph_max(graph1)
```

```
call to graph_max: {'root': [13, ['A', 'B']], 'A': [77, ['B', 'C']], 'B':  ..
.
call to node_max: root
   call to node_max: A
      call to node_max: B
      node_max returns: 88
      call to node_max: C
         call to node_max: D
         node_max returns: 42
      node_max returns: 42
   node_max returns: 88
   call to node_max: B
   node_max returns: 88
node_max returns: 88
graph_max returns: 88
```

Out[18]: 88

In [19]:
```
graph_max(graph2)
```

```
call to graph_max: {'root': [13, ['A', 'B']], 'A': [77, ['B', 'C']], 'B':  ..
.
call to node_max: root
   call to node_max: A
      call to node_max: B
      node_max returns: 88
      call to node_max: C
         call to node_max: D
         node_max returns: 42
      node_max returns: 42
   node_max returns: 88
   call to node_max: B
   node_max returns: 88
node_max returns: 88
graph_max returns: 88
```

Out[19]: 88

## 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 [20]:
```
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])
```

```
x: [0, [...], 2]
x[1][1][1][1][1][1][1][1][1][1][2]: 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 [21]:
```
@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 [22]:
```
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])
```

```
x: [0, [...], 2]
y: ['zero', [...], 2]
y[1][1][1][1][1][1][1][1][1][1][2]: 2

call to deep_copy_list: [0, [...], 2]
   call to deep_copy_list: 0, {140016068860680: []}
   deep_copy_list returns: 0
   call to deep_copy_list: [0, [...], 2], {140016068860680: [0], 10942976: 0}
   deep_copy_list returns: [0]
   call to deep_copy_list: 2, {140016068860680: [0, [...]], 10942976: 0}
   deep_copy_list returns: 2
deep_copy_list returns: [0, [...], 2]
```