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

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

```
call to walk_list: [1, 2, 3], <function <lambda> at 0x00000000056FAF28>
   call to walk_list: [2, 3], <function <lambda> at 0x00000000056FAF28>
      call to walk_list: [3], <function <lambda> at 0x00000000056FAF28>
      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
walk_list_filter1([1, 2, 3], lambda x: x % 2 == 1)
```

```
call to walk_list_filter1: [1, 2, 3], <function <lambda> at 0x00000000056FAC80>
   call to walk_list_filter1: [2, 3], <function <lambda> at 0x00000000056FAC80>
      call to walk_list_filter1: [3], <function <lambda> at 0x00000000056FAC80>
         call to walk_list_filter1: [], <function <lambda> at 0x00000000056FAC80>
         walk_list_filter1 returns: []
      walk_list_filter1 returns: [3]
   walk_list_filter1 returns: [3]
walk_list_filter1 returns: [1, 3]
```

Out[6]: [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]:
```python
walk_list_filter2([1, 2, 3], lambda x: x % 2 == 1)
```

```
call to walk_list_filter2: [1, 2, 3], <function <lambda> at 0x000000000572D950>
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]
```

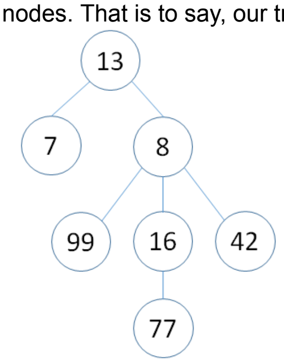
Out[8]:
```
[1, 3]
```

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 [9]:
```python
tree1 = [13,
         [7],
         [8,
          [99],
          [16,
           [77]],
          [42]]]
tree1
```

Out[9]:
```
[13, [7], [8, [99], [16, [77]], [42]]]
```

In [10]:
```python
@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
    """
    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 [11]:
```python
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[11]: 99

In [12]:
```python
@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 [13]:
```
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[13]: 4

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...

In [14]:
```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])
    [1, [2], [3]]
    >>> make_tree([1,2])
    [1, [2]]
    """
    n = len(L)
    if n == 0:          #base case
        return []

    val = L[0]
    if n == 1:          #another base case -- no children
        return [val]

    split = (n-1) // 2
    left = make_tree(L[1:split+1]) #recursive left half of list
    right = make_tree(L[split+1:]) #recursive right half of list

    #return [val, left, right]
    # FIX: left branch might be empty (right branch will never be), so
    # only combine if left is not empty:
    return [val, left, right] if left else [val, right]
```

In [15]:
```python
tree2 = make_tree([1, 2, 3])
tree2
```

```
call to make_tree: [1, 2, 3]
   call to make_tree: [2]
   make_tree returns: [2]
   call to make_tree: [3]
   make_tree returns: [3]
make_tree returns: [1, [2], [3]]
```

Out[15]:
```
[1, [2], [3]]
```

In [16]:
```python
tree3 = make_tree([1, 2]) #unbalanced tree case
tree3
```

```
call to make_tree: [1, 2]
   call to make_tree: []
   make_tree returns: []
   call to make_tree: [2]
   make_tree returns: [2]
make_tree returns: [1, [2]]
```

Out[16]:
```
[1, [2]]
```

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

In [17]:
```python
instrument.SHOW_CALL = True
instrument.SHOW_RET = False
```

In [18]:
```python
tree4 = make_tree(list(range(8)))
tree4
```

```
call to make_tree: [0, 1, 2, 3, 4, 5, 6, 7]
   call to make_tree: [1, 2, 3]
      call to make_tree: [2]
      call to make_tree: [3]
   call to make_tree: [4, 5, 6, 7]
      call to make_tree: [5]
      call to make_tree: [6, 7]
         call to make_tree: []
         call to make_tree: [7]
```

Out[18]:
```
[0, [1, [2], [3]], [4, [5], [6, [7]]]]
```

Another recursive example -- a printed visualization of a tree

In [19]:
```python
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 [20]:
```python
print("tree4:", tree4, "\n", show_tree(tree4))
```

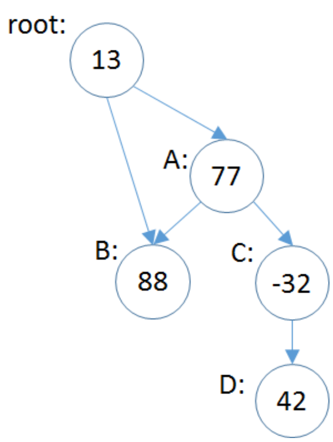
```
tree4: [0, [1, [2], [3]], [4, [5], [6, [7]]]]
 0
   1
      2
      3
   4
      5
      6
         7


call to helper: [0, [1, [2], [3]], [4, [5], [6, [7]]]], 0
   call to helper: [1, [2], [3]], 1
      call to helper: [2], 2
      call to helper: [3], 2
   call to helper: [4, [5], [6, [7]]], 1
      call to helper: [5], 2
      call to helper: [6, [7]], 2
         call to helper: [7], 3
```

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 [21]:
```python
graph1 = {'root': [13, ['A', 'B']],
          'A': [77, ['B', 'C']],
          'B': [88, []],
          'C': [-32, ['D']],
          'D': [42, []]}
```

In [22]:
```python
@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):
        val = graph[node_name][0]
        children = graph[node_name][1]
        if children:
            return max(val, max(node_max(child) for child in children))
        return val
    return node_max('root')
```

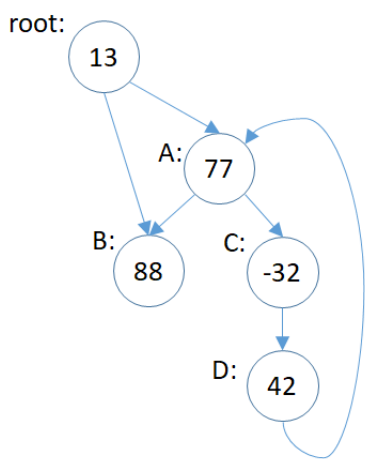
In [23]:
```python
instrument.SHOW_CALL = True
instrument.SHOW_RET = True
```

In [24]:
```python
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[24]: 88

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



In [25]:
```python
graph2 = {'root': [13, ['A', 'B']],
          'A': [77, ['B', 'C']],
          'B': [88, []],
          'C': [-32, ['D']],
          'D': [42, ['A']]} #changed; now D -> A
```

In [26]:
```python
#graph_max(graph2) # breaks (infinite recursion)! (need to re-execute def graph_max afterwards
 for instrumentation)
```

In [27]:
```python
@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 [28]:
```python
graph_max2(graph2)
```

```
call to graph_max2: {'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_max2 returns: 88
```

Out[28]: 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 [29]:
```python
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 [30]:

```python
@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 [31]:

```python
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, {93714504: []}
   deep_copy_list returns: 0
   call to deep_copy_list: [0, [...], 2], {93714504: [0], 492989440: 0}
   deep_copy_list returns: [0]
   call to deep_copy_list: 2, {93714504: [0, [...]], 492989440: 0}
   deep_copy_list returns: 2
deep_copy_list returns: [0, [...], 2]
```