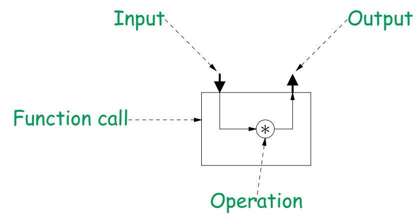
Lecture #20: Recursive Processes, Memoization, Tree Structures

Varieties of Recursive Processes

- We can characterize (potentially) recursive functions according to the patterns in which data flows through them.
- The simplest case is a non-recursive function call, which does something (call it h) to its input data and returns the result:

```
def func0(x):
    return h(x)
```

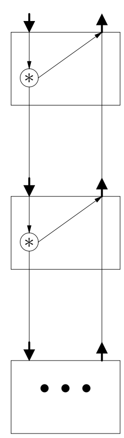


- "Operations" include any processing that does not cause further recursion.
- This is a *leaf call*.

Iterative (Tail-Recursive) Processes

- Tail-recursive processes do no further processing after a recursive call

```
def func1(x):
    if P(x):
        return h1(x)
    else:
        return func1(h2(x))
```

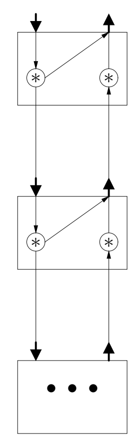


- Once we make a recursive call, can forget about the caller.
- Constant space needed for administrative overhead (in principle)
- Time required (number of operations) proportional to call depth.

Linear Recursions

- Linear recursions do one recursive call and then additional processing

```
def func2(x):
    if P(x):
        return h1(x)
    else:
        return h3((func2(h2(x)))
```

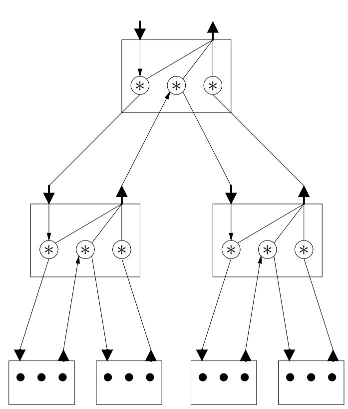


- Must keep track of pending calls, because there is more to do for each.
- Space proportional to depth of calls needed for administrative overhead.
- Time required proportional to call depth.

Tree (General) Recursion

- Tree recursions do more than one recursive call in each function execution.

```
def func3(x):
    if P1(x):
        return h1(x)
    else:
        y = func3(h2(x))
        if P2(x):
           return h3(x, y)
        z = func3(h4(x, y))
        return h5(x, y, z)
```



- Again, must keep track of pending calls (one per level).
- So, space proportional to depth of calls.
- But time required may be *exponential* in call depth.

Avoiding Redundant Computation

- In the (tree-recursive) maze example, a naive search could take us in circles, resulting in infinite time.
- Hence the visited parameter in the search function.
- This parameter is intended to catch redundant computation, in which reprocessing certain arguments cannot produce anything new.
- We can apply this idea to cases of finite but redundant computation.
- For example, in count_change, we often revisit the same subproblem:
  - E.g., Consider making change for 87 cents.
  - When choose to use one half-dollar piece, we have the same subproblem as when we choose to use no half-dollars and two quarters.
- Saw an approach in Lecture #16: memoization.

Memoizing

- Idea is to keep around a table ("memo table") of previously computed values.
- Consult the table before using the full computation.
- Example: count_change:

```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    memo_table = {}
    # Local definition hides outer one so we can cut-and-paste
    # from the unmemoized (red) solution.
    def count_change(amount, coins):
        if (amount, coins) not in memo_table:
             memo_table[amount,coins]
                = full_count_change(amount, coins)
        return memo_table[amount,coins]
    def full_count_change(amount, coins):
        original solution goes here verbatim
    return count_change(amount,coins)
```

- Question: how could we test for infinite recursion?

Optimizing Memoization

- Used a dictionary to memoize count_change, which is highly general, but can be relatively slow.
- More often, we use arrays indexed by integers (lists in Python), but the idea is the same.
- For example, in the count_change program, we can index by amount and by the portion of coins that we use, which is always a slice that runs to the end.

```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    # memo_table[amt][k] contains the value computed for
    #   count_change(amt, coins[k:])
    memo_table = [ [-1] * (len(coins)+1) for i in range(amount+1) ]
    def count_change(amount, coins):
        if memo_table[amount][len(coins)] == -1:
             memo_table[amount][len(coins)]
                = full_count_change(amount, coins)
        return memo_table[amount][len(coins)]
    ...
```


Order of Calls

- Going one step further, we can analyze the order in which our program ends up filling in the table.
- So consider adding some tracing to our memoized count_change program:

```
memo_table = {}
def count_change(amount, coins):
    ... full_count_change(amount, coins) ...
    return memo_table[amount,coins]
@trace
def full_count_change(amount, coins):
    if amount == 0: return 1
    elif not coins: return 0
    elif amount >= coins[0]:
        return count_change(amount, coins[1:]) \
               + count_change(amount-coins[0], coins)
    else:
        return count_change(amount, coins[1:])
return count_change(amount,coins)
```


Result of Tracing

- Consider count_change(57) (returns only):

```
full_count_change(57, ()) -> 0
full_count_change(56, ()) -> 0
...
full_count_change(1, ()) -> 0
full_count_change(0, (1,)) -> 1
full_count_change(1, (1,)) -> 1
...
full_count_change(57, (1,)) -> 1
full_count_change(2, (5, 1)) -> 1
full_count_change(7, (5, 1)) -> 2
...
full_count_change(57, (5, 1)) -> 12
full_count_change(7, (10, 5, 1)) -> 2
full_count_change(17, (10, 5, 1)) -> 6
...
full_count_change(32, (10, 5, 1)) -> 16
full_count_change(7, (25, 10, 5, 1)) -> 2
full_count_change(32, (25, 10, 5, 1)) -> 18
full_count_change(57, (25, 10, 5, 1)) -> 60
full_count_change(7, (50, 25, 10, 5, 1)) -> 2
full_count_change(57, (50, 25, 10, 5, 1)) -> 62
```


Dynamic Programming

- Now rewrite count_change to make the order of calls explicit, so that we needn't check to see if a value is memoized.
- Technique is called *dynamic programming* (for some reason).
- We start with the base cases, and work backwards.

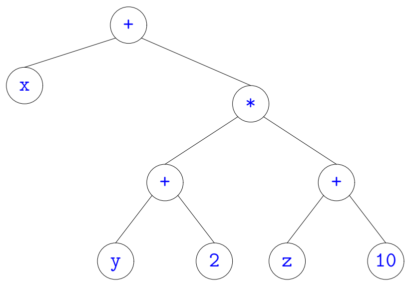
```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    memo_table = [ [-1] * (len(coins)+1) for i in range(amount+1) ]
    def count_change(amount, coins):
        return memo_table[amount][len(coins)]
    def full_count_change(amount, coins):
        # How often is this called?
        ... # (calls count_change for recursive results)

    for a in range(0, amount+1):
        memo_table[a][0] = full_count_change(a, ())
    for k in range(1, len(coins) + 1):
        for a in range(1, amount+1):
            memo_table[a][k] = full_count_change(a, coins[-k:])
    return count_change(amount, coins)
```


New Topic: Tree-Structured Data

- 1 Linear-recursive and tail-recursive functions make a single recursive call in the function body. Tree-recursive functions can make more.
- Linear recursive data structures (think rlists) have single embedded recursive references to data of the same type, and usually correspond to linear- or tail-recursive programs.
- To model some things, we need mulitple recursive references in objects.
- In the absence of circularity (paths from an object eventually leading back to it), such objects form data structures called *trees*:
  - The objects themselves are called *nodes* or *vertices*.
  - Tree objects that have no (non-null) pointers to other tree objects are called *leaves*.
  - Those that do have such pointers are called *inner nodes*, and the objects they point to are *children* (or *subtrees* or (uncommonly) *branches*).
  - A collection of disjoint trees is called a *forest*.

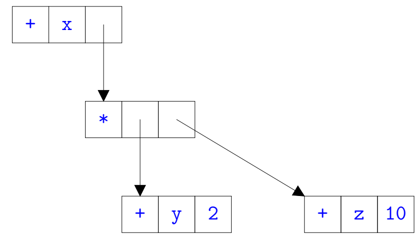
Example: Expressions

- An expression (in Python or other languages) typically has a recursive structure. It is either
  - A literal (like 5) or symbol (like x)—a leaf—or
  - A compound expression consisting of an operator and zero or more operands, each of which is itself an expression.
- For example, the expression x + (y+2)*(z+10) can be thought of as a tree (what happened to the parentheses?):



Expressions as Tuples or Lists

- We can represent the abstract structure of the last slide with Python objects we've already seen:




```
("+", "x", ("*", ("+", "y", "2"), ("+", "z", "10")))
```


Class Representation

- ...or we can introduce a Python class:

```
class ExprTree:
    def __init__(self, operator):
        self.__operator = operator

    @property
    def operator(self):
        return self.__operator

    @property
    def left(self):
        raise NotImplementedError

    @property
    def right(self):
        raise NotImplementedError
```

```
class Leaf(ExprTree):
    pass

class Inner(ExprTree):
    def __init__(self, operator,
                 left, right):
        ExprTree.__init__(self, operator)
        self.__left = left;
        self.__right = right
    @property
    def left(self):
        return self.__left
    @property
    def right(self):
        return self.__right
```

```
Inner("+", Leaf("x"),
      Inner("*", Inner("+", Leaf("y"), Leaf("2")),
                 Inner("+", Leaf("z"), Leaf("10"))))
```