Introduction to Recursion

A definition of a thing is *recursive* if its meaning includes the thing. In other words, *recursion* occurs when a thing is defined in terms of itself.

For example, the non-negative powers of 2 can be defined recursively as follows:

$2^0 = 1$
$2^k = 2*2^{k-1}$ for $k > 0$

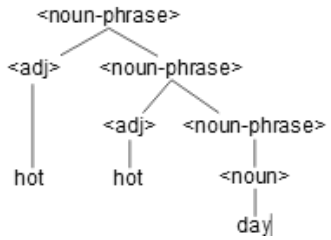
To the right, we show how this definition can be used to calculate 2^2. The calculation shows two uses of the definition in the case $k > 0$ and one use of the definition of 2^0.

2^2
= <by definition of 2^k with k = 2>
$2*2^1$
= <by definition of 2^k with k = 1>
$2*(2*2^0)$
= <by definition of 2^k with k = 0>
2*2*1
= <arithmetic>
4

Below we give a *grammar* for noun phrases, which could be part of a grammar that defines the syntax of English. Line (1) says that `dog` and `day` are nouns. The symbol "::=" is used simply to separate the term being defined from its definitions(s). In the same way, line (2) defines three adjectives. Line (3) defines a noun phrase to be either a noun or an adjective followed by a noun phrase. Aha! A recursive definition.

(1) `<noun> ::= dog | day`
(2) `<adj> ::= hot | nice | sunny`
(3) `<noun-phrase> ::= <noun> | <adj> <noun-phrase>`

To the right, we give a "tree" that uses this grammar for noun phrases to show that `hot hot day` is a noun phrase. At the top, you see the use of the recursive definition "An adjective followed by a noun phrase is a noun phrase". You can see that definition used a second time. You see one use of the definition "a noun is a noun phrase", one use of the definition "`day` is a noun", and two uses of the definition "`hot` is an adjective".



Each time the recursive definition in line (3) is used, another adjective is added. Thus, a noun phrase can have 0 or more adjectives, and the same adjective can appear over and over in a noun phrase. A grammar defines syntax, not semantics.

Here's one more recursive definition, of the set of ancestors of a person `p`:

`p`'s ancestors consist of (1) `p`'s parents and (2) the ancestors of `p`'s parents

Two recursive functions

Above, we gave a definition of the nonnegative powers of 2. Such a mathematical definition can be transformed easily, almost automatically, into a Java function, as shown to the right. You can write many recursive mathematical definitions as Java functions in this fashion.

```
/** = 2^k.
  * Precondition k >= 0. */
public static int pow(int k) {
  if (k == 0) return 1;
  return 2 * pow(k-1);
}
```

To the right below is another recursive function, which returns the number of digits in the decimal representation of a nonnegative integer `n`.

If `n` < 10, the answer is 1 (even if `n` is 0).

The comments in the function tell you that for `n` ≥ 10, the answer is 1 plus the number of digits in `n` /10. That's the value that the return statement returns. We can calculate the call numDigits(352):

numDigits(352)
= <with n = 352, the value of numDigits(n/10) + 1 is returned>
numDigits(35) + 1
= <with n = 35, the value of numDigits(n/10) + 1 is returned>
numDigits(3) + 1 + 1
= <with n = 3, the value 1 is returned>
1 + 1 + 1
= <arithmetic>
3

```
/** = number of digits in the decimal
  *   representation of n.
  *   e.g. numDigits(0) = 1,
  *        numDigits(35) = 2,
  *        numDigits(1356) = 4.
  *  Precondition: n >= 0. */
public static int numDigits(int n) {
  if (n < 10) return 1;
  // n = (n/10)*10  + n%10
  // So, #digits in n is #(n/10) + 1
  return numDigits(n/10) + 1;
}
```

Introduction to Recursion

Two ways to think about recursive methods

With a recursive method, we have two different questions, with two totally different answers:

1. How is a call on a recursive method *executed* —how could it possible work if it calls itself?
2. How do we *understand* a recursive method, and how do we *write/develop* a recursive method?

Our first task is to show you how recursive calls are executed, so that you can then know that they work. After that, we can study how to understand a recursive method and how to go about developing one.

For the first task, we ask that you visit the JavaHyperText and click the link

```
Explain constructs/3. Method calls
```

in the top horizontal navigation bar. This tutorial covers these topics:

1. The frame for a method call.
2. Two data structures: the queue and the stack.
3. The call stack and the algorithm for executing *any* method call.
4. Verification that the process for executing a method call works for recursive calls.

For the second task, look at item 3 under entry "recursion" in the JavaHyperText.