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.

22  
= <by definition of 2k with k = 2>

2\*21

= <by definition of 2k with k = 1>

2\*(2\*20)

= <by definition of 2k with k = 0>

2\*2\*1  
= <arithmetic>

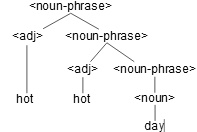
4

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

20 = 1  
2k = 2\*2k-1.  for k > 0

To the right, we show how this definition can be used to calculate 22. The calculation shows two uses of the definition in the case k > 0 and one use of the definition of 20.

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

**Writing recursive functions**

/\*\* = 2^k.

\* Precondition k >= 0. \*/

**public** **static** **int** pow(**int** k) {

**if** (k == 0) **return** 1;  
 **return** 2 \* pow(k-1);

}

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 almost any recursive mathematical definition into a Java function in this fashion.

To show you what recursive functions may look like, we present on the right below a function that returns the number of digits in the decimal representation of a number n.

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

/\*\* = 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;

}

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. Ae can calculate 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