Recursion

Let's first state the idea of recursion, and then try to make its idea concrete with a set of examples.

> Recursion is the idea that we can solve a big problem by solving a slightly smaller version of the same problem.

That's it. Time to get concrete. Or, at least, pavement. How do we climb atop some stairs? Of course we can't jump directly from the bottom to the top. We take one step up, or two if we're very daring. And then we ask the question again, how can we climb up these stairs? Well, let's take another step, literally.

And now we ask ourselves the same questions: how do we climb the stairs? Well, take a step, and try again. When we're at the top of the staircase, we don't need to climb anymore. In pseudo-Scheme, we're doing something like this:

```
(define (climb-staircase n-steps)
    (if (= n-steps 0) 'done
        (one-more-step (climb-staircase (- 1 n-steps)))))
```

Usually, we take the solution to a smaller problem and combine it with something to make it the perfect solution. For example, if we know how to climb up 5 stairs, we know how to climb up 6 stairs: just climb those 5 steps, and take another step!

Now after skillfully mounting those steps, let's try something else. Let's sing the bottles of beer chant. (I dunno, maybe we're climbing the steps to a bar or something...)

How do we sing the beers on the wall song for 2 bottles?

> 2 bottles of beer on the wall, 2 bottles of beer, take one down, pass it around, 1 bottles of beer on the wall.
>
> ... etc.

How do we sing the beers on the wall song for 3 bottles?

> 3 bottles of beer on the wall, 3 bottles of beer, take one down, pass it around, 2 bottles of beer on the wall.
>
> 2 bottles of beer on the wall, 3 bottles of beer, take one down, pass it around, 1 bottles of beer on the wall.
>
> ... etc.

So in knowing how to sing for 2 bottles, it doesn't take too much brainpower to sing it for 3 bottles: just add the beginning chant. When we run out of beer, we stop.

Let's write a program that sings that jolly song: we'd like to ask the computer:

```
(chant-beer-song 4)
```

and get back the list:

```
( (4 bottles of beer)
  (3 bottles of beer)
  (2 bottles of beer)
  (1 bottle of beer) )
```

As a small utility, let's define:

```
(define (n-bottles-of-beer n)
  (if (= n 1)
        (list n 'bottle 'of 'beer)
        (list n 'bottles 'of 'beer)))
```

How do we go about this problem? Where do we start? Well, if we have one beer, we just sing the first chant and stop.

```
(define (chant-beer-song n)
    (if (= n 1) (list (n-bottles-of-beer 1))
          'fix-me-later))
```

Let's test this:

```
STk> (chant-beer-song 1)
((1 bottle of beer))
STk> (chant-beer-song 2)
fix-me-later
```

Obviously, it doesn't work quite yet. But it's a beginning, because we can sing the song for one bottle. That turns out to be enough to make the problem work!

```
(define (chant-beer-song n)
    (if (= n 1) (list (n-bottles-of-beer 1))
          (cons (say-beer n) (chant-beer-song (- n 1)))))
```

Let's test it:

```
STk> (chant-beer-song 5)
((5 bottles of beer) (4 bottles of beer)
 (3 bottles of beer) (2 bottles of beer)
 (1 bottle of beer))
```

(Note; I've indented it to make it a little easier to read.)

If you have a math background, you may recognize recursion as an application of mathematical induction: we explain to Scheme what our "base" cases

are. Once we've done that, we assume that things work for m, and then use that to make it work for $m+1$. An example of math induction is the following:

Prove that $1 + 2 + ... + n = n(n+1)/2$ for all $n \geq 1$.

First, let's show that this works for the base case, $m = 1$:

$$1 = 1(1+1)/2 = 1.$$

Yes, this works. Now, let's assume that this works for $n = m$. Now let's show that the above works for $m + 1$:

We ask the question: is it true that:

$$1 + 2 + ... + (m+1) = (m+1)(m+2)/2 \qquad ?$$

Let's see:

$$\begin{array}{rcll} 1 + 2 + ... + (m+1) & = & (1 + 2 + ... + m) + (m+1) & \\ & = & m(m+1)/2) + (m+1) & \text{by our assumption.} \\ & = & (m+1)(m/2+1) & \\ & = & (m+1)(m+2)/2. & \text{done.} \end{array}$$

Why does this work? The idea is that we have a solid foundation, our base case, that lets us expand our knowledge of the problem. We know that the statement works for $n = 1$, and, given that it works for $n = m$, we can prove that it works for $n = m + 1$. Since we know it works for $n = 1$, we know that it works for $n = 1 + 1 = 2$. Now, since it works for $n = 2$, it works for $n = 2 + 1 = 3$. If we continue along this path, we know that this statement is true for any $n \geq 1$.

Let's relate this back to our programming. When we design our programs, we want to figure out the "base case", the simplest problem our program should handle without any effort. When we deal with lists, the base case usually deals with lists of zero or one elements.

After writing our base cases, we assume that the program, for all intensive purposes, works for small problems. Our job, then, is to make it work for the current problem. For example, how do we sum the numbers from 1 to 10? We add 10 to the sum from 1 to 9. The simplest question we can ask ourselves is: How do we sum from 1 to 1? Easy, that's just 1:

```
(define (sum-from-1 n)
  (if (= n 1) 1
    (+ n (sum-from-1 (- n 1)))))
```

And this works!

```
> (sum-from-1 1)
1
> (sum-from-1 2)
3
> (sum-from-1 10)
55
```