Linear Recursion

We will now consider functions that generate computation of varying size depending upon the input. The technique use to achieve this is called *recursion*. The algorithm breaks up the task into smaller instances of the original task and then combines the results in some way.

We will illustrate this with the task of making a paper chain. Consider of line of students and pile of paper strips and a stapler:

Make a chain of length n:

1. If $n = 1$
   (a) Bend a strip into a loop and join the ends
   (b) Deliver your chain of length 1 to your customer
2. Otherwise
   (a) Pick up a strip
   (b) As the person next to you to make you a chain of length $n - 1$
   (c) When you get your chain, slip your strip through one end and join the ends, adding another link
   (d) Deliver your chain of length n to your customer

Notice that the problem of making a chain was broken up into two cases: a base case of length 1 and a recursive case where we used the same algorithm to solve a smaller instance of the same problem (making a chain of length $n-1$) and then did some extra work on the result (linking another strip) to create our chain. This strategy is called *linearrecursion*. It is time to look at examples in Haskell.

Suppose we want to compute the number of different ordering of a deck of cards. We would have 52 possibilities for the choice of the first card, 51 possibilities for the second card, 50 possibilities for the third card, etc. Therefore the number of possible orderings would be $52*51*50*...*1$ or 52 factorial. Here is a Haskell function to compute n factorial:

```
fact :: Int -> Int
fact n = if n==0
               then 1
               else n * fact (n - 1)
```

Just as with the chain example, the definition is broken up into a base case, where n is 0, and a recursive case which uses a smaller instance of the problem, $fact(n-1)$. We can use the substitution model to examine the computation of $fact$ 4:

Expression	Substitution explanation
$fact\ 4$	substitute into the body of $fact$
$4 * fact\ 3$	substitute for $fact\ 3$
$4 * (3 * fact\ 2)$	substitute for $fact\ 2$
$4 * (3 * (2 * fact\ 1))$	substitute for $fact\ 1$
$4 * (3 * (2 * (1 * fact\ 0)))$	substitute for $fact\ 0$
$4 * (3 * (2 * (1 * 1)))$	multiply
$4 * (3 * (2 * 1))$	multiply
$4 * (3 * 2)$	multiply
$4 * 6$	multiply
24	

Observe that the computation is composed of a *winding* phase where the expressions are getting larger and larger with each recursive substitution, and an *unwinding* phase where the results are being combined.

Let's look at an example that uses Words. The following function reverses the letters in a word:

```
revWord :: Language -> Language
revWord w = if (empty w)
                    then w
                    else (lastItem w) +++ revWord (butLast w)
```

Use the substitution model to look at the computation `revWord (word "cat")`.

Try writing linear recursive functions for the following:

```
-- compute the base value raised to the power of the exponent
power :: Int -> Int -> Int
power base exp = ...

-- compute the number of letters in a word
length :: Language -> Int
length w = ...
```