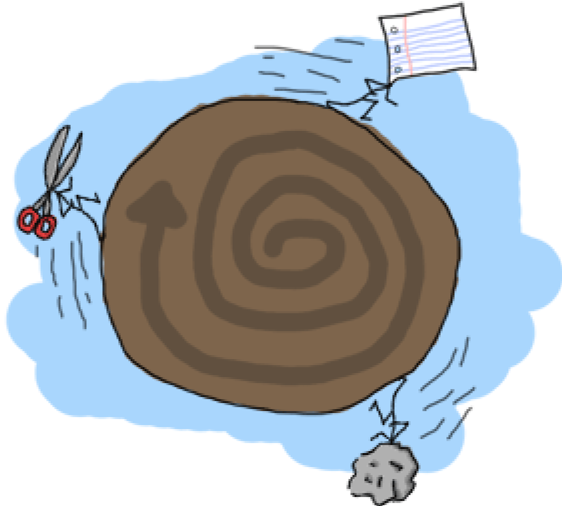
MORE RECURSIVE FUNCTIONS



We'll write a few more recursive functions, just to get in the habit a bit more. After all, recursion being the only looping construct that exists in Erlang (except list comprehensions), it's one of the most important concepts to understand. It's also useful in every other functional programming language you'll try afterwards, so take notes!

The first function we'll write will be `duplicate/2`. This function takes an integer as its first parameter and then any other term as its second parameter. It will then create a list of as many copies of the term as specified by the integer. Like before, thinking of the base case first is what might help you get going. For `duplicate/2`, asking to repeat something 0 time is the most basic thing that can be done. All we have to do is return an empty list, no matter what the term is. Every other case needs to try and get to the base case by calling the function itself. We will also forbid negative values for the integer, because you can't duplicate something `-n` times:

```
duplicate(0,_) ->
[];
duplicate(N,Term) when N > 0 ->
[Term|duplicate(N-1,Term)].
```

Once the basic recursive function is found, it becomes easier to transform it into a tail recursive one by moving the list construction into a temporary variable:

```
tail_duplicate(N,Term) ->
tail_duplicate(N,Term,[]).
```

```
tail_duplicate(0,_,List) ->
List;
tail_duplicate(N,Term,List) when N > 0 ->
tail_duplicate(N-1, Term, [Term|List]).
```

Success! I want to change the subject a little bit here by drawing a parallel between tail recursion and a while loop. Our `tail_duplicate/2` function has all the usual parts of a while loop. If we were to imagine a while loop in a fictional language with Erlang-like syntax, our function could look a bit like this:

```
function(N, Term) ->
while N > 0 ->
List = [Term|List],
N = N-1
end,
List.
```

Note that all the elements are there in both the fictional language and in Erlang. Only their position changes. This demonstrates that a proper tail recursive function is similar to an iterative process, like a while loop.

There's also an interesting property that we can 'discover' when we compare recursive and tail recursive functions by writing a `reverse/1` function, which will reverse a list of terms. For such a function, the base case is an empty list, for which we have nothing to reverse. We can just return an empty list when that happens. Every other possibility should try to converge to the base case by calling itself, like with `duplicate/2`. Our function is going to iterate through the list by pattern matching `[H|T]` and then putting H after the rest of the list:

```
reverse([]) -> [];
reverse([H|T]) -> reverse(T)++[H].
```

On long lists, this will be a true nightmare: not only will we stack up all our append operations, but we will need to traverse the whole list for every single of these appends until the last one! For visual readers, the many checks can be represented as:

```
reverse([1,2,3,4]) = [4]++[3]++[2]++[1]

                       ↑      ↲

                   = [4,3]++[2]++[1]

                       ↑  ↑      ↲

                   = [4,3,2]++[1]

                       ↑  ↑  ↑      ↲

                   = [4,3,2,1]
```

This is where tail recursion comes to the rescue. Because we will use an accumulator and will add a new head to it every time, our list will automatically be reversed. Let's first see the implementation:

```
tail_reverse(L) -> tail_reverse(L,[]).

tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).
```

If we represent this one in a similar manner as the normal version, we get:

```
tail_reverse([1,2,3,4]) = tail_reverse([2,3,4],
[1])

                        = tail_reverse([3,4],
[2,1])
```

```
                        = tail_reverse([4],
[3,2,1])

                        = tail_reverse([],
[4,3,2,1])

                        = [4,3,2,1]
```

Which shows that the number of elements visited to reverse our list is now linear: not only do we avoid growing the stack, we also do our operations in a much more efficient manner!

Another function to implement could be `sublist/2`, which takes a list L and an integer N, and returns the N first elements of the list. As an example, `sublist([1,2,3,4,5,6],3)` would return `[1,2,3]`. Again, the base case is trying to obtain 0 elements from a list. Take care however, because `sublist/2` is a bit different. You've got a second base case when the list passed is empty! If we do not check for empty lists, an error would be thrown when calling `recursive:sublist([1],2).` while we want `[1]` instead. Once this is defined, the recursive part of the function only has to cycle through the list, keeping elements as it goes, until it hits one of the base cases:

```
sublist(_,0) -> [];
sublist([],_) -> [];
sublist([H|T],N) when N > 0 -> [H|sublist(T,N-1)].
```

Which can then be transformed to a tail recursive form in the same manner as before:

```
tail_sublist(L, N) -> tail_sublist(L, N, []).

tail_sublist(_, 0, SubList) -> SubList;
tail_sublist([], _, SubList) -> SubList;
tail_sublist([H|T], N, SubList) when N > 0 ->
tail_sublist(T, N-1, [H|SubList]).
```

There's a flaw in this function. *A fatal flaw!* We use a list as an accumulator in exactly the same manner we did to reverse our list. If you compile this function as is, `sublist([1,2,3,4,5,6],3)` would not return `[1,2,3]`, but `[3,2,1]`. The only thing we can do is take the final result and reverse it ourselves. Just change the `tail_sublist/2` call and leave all our recursive logic intact:

```
tail_sublist(L, N) -> reverse(tail_sublist(L, N, [])).
```

The final result will be ordered correctly. It might seem like reversing our list after a tail recursive call is a waste of time and you would be partially right (we still save memory doing this). On shorter lists, you might find your code is running faster with normal recursive calls than with tail recursive calls for this reason, but as your data sets grow, reversing the list will be comparatively lighter.

Note: instead of writing your own `reverse/1` function, you should use `lists:reverse/1`. It's been used so much for tail recursive calls that the maintainers and developers of Erlang decided to turn it into a BIF. Your lists can now benefit from extremely fast reversal (thanks to functions written in C) which will make the reversal disadvantage a lot less obvious. The rest of the code in this chapter will make use of our own reversal function, but after that you should not use it ever again.

To push things a bit further, we'll write a zipping function. A zipping function will take two lists of same length as parameters and will join them as a list of tuples which all hold two terms. Our own `zip/2` function will behave this way:

```
1> recursive:zip([a,b,c],[1,2,3]).
[{a,1},{b,2},{c,3}]
```

Given we want our parameters to both have the same length, the base case will be zipping two empty lists:

```
zip([],[]) -> [];
zip([X|Xs],[Y|Ys]) -> [{X,Y}|zip(Xs,Ys)].
```

However, if you wanted a more lenient zip function, you could decide to have it finish whenever one of the two list is done. In this scenario, you therefore have two base cases:

```
lenient_zip([],_) -> [];
```

```
lenient_zip(_,[]) -> [];
lenient_zip([X|Xs],[Y|Ys]) ->
[{X,Y}|lenient_zip(Xs,Ys)].
```

Notice that no matter what our base cases are, the recursive part of the function remains the same. I would suggest you try and make your own tail recursive versions of `zip/2` and `lenient_zip/2`, just to make sure you fully understand how to make tail recursive functions: they'll be one of the central concepts of larger applications where our main loops will be made that way.

If you want to check your answers, take a look at my implementation of recursive.erl, more precisely the`tail_zip/2` and `tail_lenient_zip/3` functions.

Note: tail recursion as seen here is not making the memory grow because when the virtual machine sees a function calling itself in a tail position (the last expression to be evaluated in a function), it eliminates the current stack frame. This is called tail-call optimisation (TCO) and it is a special case of a more general optimisation named *Last Call Optimisation* (LCO).

LCO is done whenever the last expression to be evaluated in a function body is another function call. When that happens, as with TCO, the Erlang VM avoids storing the stack frame. As such tail recursion is also possible between multiple functions. As an example, the chain of functions `a() -> b(). b() -> c(). c() -> a().` will effectively create an infinite loop that won't go out of memory as LCO avoids overflowing the stack. This principle, combined with our use of accumulators is what makes tail recursion useful.

Source : http://learnyousomeerlang.com/recursion