That's all very well, but how does this help us? Most of the time, it's a convenience function so that we don't have to write so many parentheses. Consider the expression `sum (map sqrt [1..130])`. Because `$` has such a low precedence, we can rewrite that expression as `sum $ map sqrt [1..130]`, saving ourselves precious keystrokes! When a `$` is encountered, the expression on its right is applied as the parameter to the function on its left. How about `sqrt 3 + 4 + 9`? This adds together 9, 4 and the square root of 3. If we want get the square root of *3* + *4* + *9*, we'd have to write `sqrt (3 + 4 + 9)` or if we use `$` we can write it as `sqrt $ 3 + 4 + 9` because `$` has the lowest precedence of any operator. That's why you can imagine a `$` being sort of the equivalent of writing an opening parentheses and then writing a closing one on the far right side of the expression.

How about `sum (filter (> 10) (map (*2) [2..10]))`? Well, because `$` is right-associative, `f (g (z x))` is equal to `f $ g $ z x`. And so, we can rewrite `sum (filter (> 10) (map (*2) [2..10]))` as `sum $ filter (> 10) $ map (*2) [2..10]`.

But apart from getting rid of parentheses, `$` means that function application can be treated just like another function. That way, we can, for instance, map function application over a list of functions.

```
ghci> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

Function composition

In mathematics, function composition is defined like this: $(f \circ g)(x) = f(g(x))$, meaning that composing two functions produces a new function that, when called with a parameter, say, *x* is the equivalent of calling *g* with the parameter *x* and then calling the *f* with that result.

In Haskell, function composition is pretty much the same thing. We do function composition with the `.` function, which is defined like so:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Mind the type declaration. `f` must take as its parameter a value that has the same type as `g`'s return value. So the resulting function takes a parameter of the same type that `g` takes and returns a value of the same type that `f` returns. The expression `negate . (* 3)` returns a function that takes a number, multiplies it by 3 and then negates it.

One of the uses for function composition is making functions on the fly to pass to other functions. Sure, can use lambdas for that, but many times, function composition is clearer and more concise. Say we have a list of numbers and we want to turn them all

into negative numbers. One way to do that would be to get each number's absolute value and then negate it, like so:

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Notice the lambda and how it looks like the result function composition. Using function composition, we can rewrite that as:

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Fabulous! Function composition is right-associative, so we can compose many functions at a time. The expression `f (g (z x))` is equivalent to `(f . g . z) x`. With that in mind, we can turn

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

into

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

But what about functions that take several parameters? Well, if we want to use them in function composition, we usually have to partially apply them just so much that each function takes just one parameter. `sum (replicate 5 (max 6.7 8.9))` can be rewritten as `(sum . replicate 5 . max 6.7) 8.9` or as `sum . replicate 5 . max 6.7 $ 8.9`. What goes on in here is this: a function that takes what `max 6.7` takes and applies `replicate 5` to it is created. Then, a function that takes the result of that and does a sum of it is created. Finally, that function is called with `8.9`. But normally, you just read that as: apply `8.9` to `max 6.7`, then apply `replicate 5` to that and then apply `sum` to that. If you want to rewrite an expression with a lot of parentheses by using function composition, you can start by putting the last parameter of the innermost function after a `$` and then just composing all the other function calls, writing them without their last parameter and putting dots between them. If you have `replicate 100 (product (map (*3) (zipWith max [1,2,3,4,5] [4,5,6,7,8])))`, you can write it as `replicate 100 . product . map (*3) . zipWith max [1,2,3,4,5] $ [4,5,6,7,8]`. If the expression ends with three parentheses, chances are that if you translate it into function composition, it'll have three composition operators.

Another common use of function composition is defining functions in the so-called point free style (also called the point*less* style). Take for example this function that we wrote earlier:

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
```

The `xs` is exposed on both right sides. Because of currying, we can omit the `xs` on both sides, because calling `foldl (+) 0`

creates a function that takes a list. Writing the function as `sum' = foldl (+) 0` is called writing it in point free style. How would we write this in point free style?

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

We can't just get rid of the `x` on both right right sides. The `x` in the function body has parentheses after it. `cos (max 50)` wouldn't make sense. You can't get the cosine of a function. What we can do is express `fn` as a composition of functions.

```
fn = ceiling . negate . tan . cos . max 50
```

Excellent! Many times, a point free style is more readable and concise, because it makes you think about functions and what kind of functions composing them results in instead of thinking about data and how it's shuffled around. You can take simple functions and use composition as glue to form more complex functions. However, many times, writing a function in point free style can be less readable if a function is too complex. That's why making long chains of function composition is discouraged, although I plead guilty of sometimes being too composition-happy. The prefered style is to use *let* bindings to give labels to intermediary results or split the problem into sub-problems and then put it together so that the function makes sense to someone reading it instead of just making a huge composition chain.

In the section about maps and filters, we solved a problem of finding the sum of all odd squares that are smaller than 10,000. Here's what the solution looks like when put into a function.

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

Being such a fan of function composition, I would have probably written that like this:

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
```

However, if there was a chance of someone else reading that code, I would have written it like this:

```
oddSquareSum :: Integer
oddSquareSum =
    let oddSquares = filter odd $ map (^2) [1..]
        belowLimit = takeWhile (<10000) oddSquares
    in  sum belowLimit
```

It wouldn't win any code golf competition, but someone reading the function will probably find it easier to read than a composition chain.