Intro to Haskell Notes: Part 6

Adrian Brasoveanu[*]

October 6, 2013

Contents

1 Maps and filters

1.1 *map*

map takes a function and a list and applies that function to every element in the list, producing a new list.

[*]Based primarily on *Learn You a Haskell for Great Good!* by Miran Lipovača, `http://learnyouahaskell.com/`.

Let's see its type signature:

```
ghci 1> :t map
    map :: (a → b) → [a] → [b]
```

The type signature says that it takes a function that takes an *a* and returns a *b*, then it takes a list of *a*-s and finally, it returns a list of *b*-s. We get a pretty good idea about what *map* does just by looking at its type signature.

Here's the documentation for *map*:

```
ghci 2> :! hoogle -- info map
    Prelude map :: (a -> b) -> [a] -> [b]
    map f xs is the list obtained by applying f to each element of xs, i.e.,

        map f [x1, x2, ..., xn] ≡ [f x1, f x2, ..., f xn]
        map f [x1, x2, ...] ≡ [f x1, f x2, ...]

    From package base map :: (a -> b) -> [a] -> [b]
```

And here's its actual definition:

```
(1)  map _ [ ] = [ ]
     map f (x : xs) = f x : map f xs
```

map is one of those really versatile higher order functions that can be used in many different ways:

```
ghci 3> map (+3) [1, 5, 3, 1, 6]
    [4, 8, 6, 4, 9]
```

```
ghci 4> map (++"!") ["BIFF", "BANG", "POW"]
    ["BIFF!", "BANG!", "POW!"]
```

```
ghci 5> map (replicate 3) [3 . . 6]
    [[3, 3, 3], [4, 4, 4], [5, 5, 5], [6, 6, 6]]
```

Note the iterated use of *map* below, which enables us to preserve the nested list structure in the output:

```
ghci 6> map (map (↑2)) [[1, 2], [3, 4, 5, 6], [7, 8]]
    [[1, 4], [9, 16, 25, 36], [49, 64]]
```

```
ghci 7> map fst [(1, 2), (3, 5), (6, 3), (2, 6), (2, 5)]
    [1, 3, 6, 2, 2]
```

You've probably noticed that each of these could be achieved with a list comprehension. For example, *map* (+3) [1, 5, 3, 1, 6] is the same as writing $[x + 3 \mid x \leftarrow [1, 5, 3, 1, 6]]$.

```
ghci 8> map (+3) [1,5,3,1,6]
    [4,8,6,4,9]
```

```
ghci 9> [x + 3 | x <- [1,5,3,1,6]]
    [4,8,6,4,9]
```

However, using *map* is much more readable for cases where we only apply a function to the elements of a list – especially if we're dealing with maps of maps.

1.2 Using *map* to get a list of functions

Using *map*, we can also do *map* (∗) [0 . .], which illustrates nicely how currying works and how (partially applied) functions are actual values that we can pass around to other functions or put into lists.

What happens here is that the function ∗, which is of type $(Num\ a) \Rightarrow a \to a \to a$, is applied to each number in the infinite list [0 . .]. Applying a function that takes two arguments to only one argument returns a function that takes one argument (the remaining one). So if we map ∗ over the list [0 . .], we get back a list of functions, each of which takes one argument, so they're all of type $(Num\ a) \Rightarrow [a \to a]$.

That is, *map* (∗) [0 . .] produces an infinite list of functions like the one we'd get by writing [(0∗), (1∗), (2∗), (3∗), (4∗), (5∗), ...].

```
ghci 10> let listOfFuns = map (*) [0..]
```

```
ghci 11> (listOfFuns !! 4) 5
    20
```

Getting the element with the index 4 from our list returns a function that's equivalent to (4∗). And then, we just apply that function to 5. So that's like writing (4∗) 5 or more simply, 4 ∗ 5.

1.3 *filter*

filter is a function that takes a predicate, i.e., a function that returns a boolean value, and a list and then returns the list of elements that satisfy the predicate.

Its type signature is as follows:

```
ghci 12> :t filter
    filter :: (a -> Bool) -> [a] -> [a]
```

This is the documentation for *filter*:

```
ghci 13> :! hoogle -- info filter
    Prelude filter :: (a -> Bool) -> [a] -> [a]
    filter, applied to a predicate and a list, returns the list of those elements that satisfy the
    predicate; i.e.,

        filter p xs = [x | x <- xs, p x]

    From package base filter :: (a -> Bool) -> [a] -> [a]
```

And this is its definition:

```
(2)  filter _ [] = []
     filter p (x : xs)
        | p x = x : filter p xs
        | otherwise = filter p xs
```

Thus, if $p\ x$ evaluates to *True*, the element gets included in the new list. If it doesn't, it stays out.

```
ghci 14> filter (>3) [1,5,3,2,1,6,4,3,2,1]
    [5,6,4]
```

```
ghci 15> filter (≡ 3) [1,2,3,4,5]
    [3]
```

```
ghci 16> filter even [1 .. 10]
    [2,4,6,8,10]
```

```
ghci 17> let notNull x = ¬ (null x)
         in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
    [[1,2,3],[3,4,5],[2,2]]
```

```
ghci 18> filter (∈ ['a' .. 'z']) "abcABC"
    "abc"
```

```
ghci 19> filter (∈ ['A' .. 'Z']) "abcABC"
    "ABC"
```

All of this can also be achieved with list comprehensions by using predicates. There's no set rule for when to use *map* and *filter* vs. using list comprehension, you just have to decide what's more readable in any given situation.

The *filter* equivalent of applying several predicates in a list comprehension is either filtering something several times or joining the predicates with the logical $\wedge$ function.

```
ghci 20> filter (∈ [1 .. 20]) (filter (∈ [10 .. 30]) [5, 10, 15, 20, 25, 30, 35])
    [10, 15, 20]
```

```
ghci 21> let inBothLists x = x ∈ [1 .. 20] ∧ x ∈ [10 .. 30]
         in filter inBothLists [5, 10, 15, 20, 25, 30, 35]
    [10, 15, 20]
```

2 More examples of using *map* and *filter*

2.1 Implementing quicksort again

Recall our *quicksort* function: we used list comprehensions to filter out the list elements that are smaller than or equal to the pivot on one hand, and larger than the pivot on the other. We can achieve the same functionality in a more readable way by using *filter*:

```
ghci 22> let {quicksort :: (Ord a) ⇒ [a] → [a];
         quicksort [] = [];
         quicksort (x : xs) =
            let smaller = quicksort (filter (⩽ x) xs); bigger = quicksort (filter (>x) xs)
            in smaller ++ [x] ++ bigger}
```

```
ghci 23> quicksort [1, 4, 2, 15, 2001, 1000, 4, 7]
    [1, 2, 4, 4, 7, 15, 1000, 2001]
```

Mapping and filtering is the bread and butter of every functional programmer's toolbox. It doesn't really matter if you do it with the *map* and *filter* functions or list comprehensions, just pick whichever one is more readable in a particular case.

2.2 Find the largest number under 100000 divisible by 3829

Let's find the largest number under 100000 that's divisible by 3829. To do that, we'll just filter a set of possibilities that contains the solution.

```
ghci 24> let {largestDivisible :: (Integral a) ⇒ a;
         largestDivisible = head (filter p [100000, 99999 ..])
            where p x = x 'mod' 3829 ≡ 0}
```

We first make a list of all numbers lower than 100000, descending. Then we filter it by our predicate and because the numbers are sorted in a descending manner, the largest number that satisfies our predicate is the first element of the filtered list.

```
ghci 25> largestDivisible
    99554
```

We didn't even need to use a finite list for our starting set. That's laziness in action again: we only end up using the head of the filtered list so it doesn't matter if the filtered list is finite or infinite; the evaluation stops when the first adequate solution is found.

2.3 *takeWhile* and the sum of all odd squares smaller than 10000

We are now going to find the sum of all odd squares that are smaller than 10000.

But first we're going to introduce the *takeWhile* function because we'll be using it in our solution. Here's its type:

```
ghci 26> :t takeWhile
    takeWhile :: (a → Bool) → [a] → [a]
```

takeWhile takes a predicate and a list and then goes from the beginning of the list and returns its elements while the predicate holds true. Once an element is found for which the predicate doesn't hold, it stops.

For example, if we wanted to get the first word of the string `"elephants know how to party"`, we could do:

```
ghci 27> takeWhile (≢ ' ') "elephants know how to party"
    "elephants"
```

Now let's compute the sum of all odd squares that are smaller than 10000.

- first, we'll begin by mapping the $(\uparrow 2)$ function over the infinite list $[1\,..]$
- then we filter the resulting list of squares so we only get the odd ones
- then we take elements from that list while they are smaller than 10000
- finally, we get the sum of all those elements

We can do all this in one line in `ghci`:

```
ghci 28> sum (takeWhile (<10000) (filter odd (map (↑2) [1 ..])))
    166650
```

We start with some initial data (the infinite list of all natural numbers) and then we map over it, filter it and cut it until it suits our needs. Finally, we just sum it up.

We could have also written this using list comprehensions:

```
ghci 29> sum (takeWhile (<10000) [n ↑ 2 | n ← [1 ..], odd (n ↑ 2)])
    166650
```

Just choose whichever one you find more readable.

Again, Haskell's laziness is what makes this possible: we can map over and filter an infinite list because it won't actually map and filter it right away, it'll delay those actions.

Things get computed only when we force Haskell to show us the sum. At that point, the sum function says to *takeWhile* that it needs those numbers. Then *takeWhile* forces the filtering and mapping to occur, but only until a number greater than or equal to 10000 is encountered.

2.4 Collatz sequences

Collatz sequences are formed as follows:

- we take a natural number:
  - if that number is even, we divide it by 2
  - if it's odd, we multiply it by 3 and then add 1 to that
- we take the resulting number and apply the same thing to it, which produces a new number and so on

It is conjectured (but not known) that for all starting numbers, the chains reach number 1 in a finite number of steps.

So if we take the starting number 13, we get this sequence: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. That is, $13 * 3 + 1$ equals 40. 40 divided by 2 is 20 etc. We see that the chain has 10 terms.

What we want to know is this: for all starting numbers between 1 and 100, how many chains have a length greater than 15? We first write a function that produces a chain:

```
ghci 30> let {chain :: (Integral a) => a -> [a];
              chain 1 = [1];
              chain n
                 | even n = n : chain (n `div` 2)
                 | odd n =  n : chain (n * 3 + 1)}
```

Because the chains end at 1, that's the edge case. The recursive part just implements the even and odd cases above. This is a pretty standard recursive function.

Let's try it with a couple of numbers to see if it appears to behave correctly.

```
ghci 31> chain 13
    [13,40,20,10,5,16,8,4,2,1]
```

```
ghci 32> chain 10
    [10,5,16,8,4,2,1]
```

```
ghci 33> chain 1
    [1]
```

```
ghci 34> chain 30
    [30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

And this is the function that tells us the answer to our question:

```
ghci 35> let {numLongChains :: Int;
              numLongChains = length (filter isLong (map chain [1..100]))
                where isLong xs = length xs > 15}
```

- we map the *chain* function over $[1\,..\,100]$ to get a list of chains, which are themselves represented as lists
- then we filter them by a predicate that just checks whether the length of a list is >15
- once we've done the filtering, we see how many chains are left in the resulting list

```
ghci 36> numLongChains
    66
```

The type of the *numLongChains* function is *Int* because *length* returns an *Int* instead of a *Num* for historical reasons. If we wanted to return a more general *Num a*, we could have used *fromIntegral* on the output length.

3 Lambdas

Lambdas are basically anonymous functions that are used because we need some functions only once. Normally, we make a lambda with the sole purpose of passing it to a higher order function.

To make a lambda, we write a \ because it kind of looks like the Greek letter lambda if you squint hard enough (out typesetting system will actually always display it properly as λ) and then we write the parameters, separated by spaces. After that comes a $\rightarrow$ and then the function body. We usually put all this in parentheses because otherwise the scope of the λ extends all the way to the right.

Here are a couple of simple examples:

```
ghci 37> (λx → x + 3) 7
    10
```

```
ghci 38> (λs → s ++ "man") "Bat"
    "Batman"
```

```
ghci 39> (λxs → "john" ∈ xs) ["john","bill","mary","liz"]
    True
```

```
ghci 40> (λxs → "john" ∈ xs) ["bill","mary","liz"]
    False
```

Just like the syntax of types, the syntax of λ-expressions in Montagovian semantics is slightly different, e.g., we often use a dot . instead of an arrow $\rightarrow$ to separate the variable(s) bound by λ-abstraction from the expression that we are abstracting over. But both for types and for lambdas, the differences are purely cosmetic: the underlying logical system and its semantics are exactly the same.

3.1 Collatz sequences again

We used a **where** binding in our *numLongChains* function to make the *isLong* function for the sole purpose of passing it to *filter*. Well, instead of doing that, we can use a lambda:

```
ghci 41> let {numLongChains :: Int;
             numLongChains = length (filter (λxs → length xs > 15) (map chain [1 . . 100]))}
```

```
ghci 42> numLongChains
    66
```

Lambdas are expressions, that's why we can pass them freely to other functions like that. Being an 'expression' means being a syntactic construct that has a semantic value on its own; **where** bindings, for example, are not expressions.

The λ-expression $\lambda xs \rightarrow length\ xs > 15$ denotes / has as its semantic value a predicate that tells us whether the length of the list passed to it is greater than 15.

3.2 η-reduction

If you are not well acquainted with currying and partial application, you might use lambdas where you don't really need to.

For example, the expressions:

$$map\ (+3)\ [1, 6, 3, 2]$$

and

$$map\ (\lambda x \rightarrow x + 3)\ [1, 6, 3, 2]$$

are equivalent since both $(+3)$ and $(\lambda x \rightarrow x + 3)$ are functions that take a number and add 3 to it.

Using a lambda in this case just makes the code more verbose:

```
ghci 43> map (+3) [1, 6, 3, 2]
    [4, 9, 6, 5]
```

```
ghci 44> map (λx → x + 3) [1, 6, 3, 2]
    [4, 9, 6, 5]
```

Reducing the more complicated expression $\lambda x \rightarrow x + 3$ to the simpler one $(+3)$ is an instance of the rule of η-reduction in λ-calculus.

There are two other term reduction rules, namely α-reduction and β-reduction, in addition to η-reduction. All of these rules are sound (when properly defined) in the sense that the term resulting from the reduction has the same denotation as the initial term:

- α-reduction is just a rule licensing alphabetical variants; for example, $\lambda x \rightarrow x + 3$ can be α-reduced to the equivalent term $\lambda y \rightarrow y + 3$
- β-reduction is just function application: $(\lambda x \rightarrow x + 3)\ 7$ is β-reduced to $7 + 3$ (which then evaluates to 10)

Here's another example of η-reduction. The two function definitions below are equivalent:

```
ghci 45> let {addThree :: (Num a) ⇒ a → a → a → a;
              addThree x y z = x + y + z}
```

```
ghci 46> let {addThree :: (Num a) ⇒ a → a → a → a;
              addThree = λx → λy → λz → x + y + z}
```

If we define the *addThree* function as we did the second time around, i.e., with three explicit λ-abstractions, it's obvious why the type declaration is what it is: there are three instances of $\rightarrow$ in both the type declaration and the actual function definition.

A lot of meaning definitions in natural language semantics use this 'explicit λ-abstraction' style. For example, instead of simply writing

$$\textsc{love}$$

we write

$$\lambda y.\, \lambda x.\, \textsc{love}\, y\, x$$

to explicitly indicate the meaning combinatorics.

Usually, we manipulate the formula we abstract over even further to make it look more like first-order logic, and we write:

$$\lambda y.\, \lambda x.\, \textsc{love}(x, y)$$

But the different ways of writing the meaning of the English verb *love* are equivalent by two applications of η-reduction:

$$\lambda y.\, \lambda x.\, \textsc{love}\, y\, x$$

is equivalent to

$$\lambda y.\, \textsc{love}\, y$$

which is equivalent to

$$\textsc{love}.$$

While the longer way of writing meanings with explicit λ-abstractions makes the combinatorics clear, the shorter way is usually more readable in longer semantic derivations or programs.

For example, the first definition of the function *addThree* that we listed above actually provides an excellent compromise between the two: $addThree\ x\ y\ z = x + y + z$ both indicates the combinatorics of the function and gives its definition in a concise way.

3.3 λ-abstracting over multiple parameters

Like normal functions, lambdas can take any number of parameters:

```
ghci 47> zipWith (λa b → (a ∗ 30 + 3) / b) [5, 4, 3, 2, 1] [1, 2, 3, 4, 5]
    [153.0, 61.5, 31.0, 15.75, 6.6]
```

3.4 Pattern matching with lambdas

And like normal functions, you can pattern match in lambdas. The only difference is that you can't define several patterns for one parameter, like making a $[\,]$ and a $(x : xs)$ pattern for the same parameter and then having values fall through.

A runtime error occurs if pattern matching fails in a lambda, so we should use it with care.

```
ghci 48> map (λ(a,b) → a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
    [3,8,9,8,7]
```

```
ghci 49> map (λ(a,b) → a + b) [(1,2,2),(3,5,5),(6,3,3),(2,6,6),(2,5,5)]
```

```
ghci 50> map (λ(a,b,c) → a + b) [(1,2,2),(3,5,5),(6,3,3),(2,6,6),(2,5,5)]
    [3,8,9,8,7]
```

Again, lambdas are normally surrounded by parentheses unless we want their scope to extend all the way to the right.

3.5 Implementing *flip* again

As we already indicated, there are times when using lambda notation makes things more transparent rather than less transparent, e.g., when we want to emphasize the combinatorics of an expression.

Because of this, the *flip* ('passivization') function is probably best defined as shown below:

```
ghci 51> let {flip' :: (a → b → c) → b → a → c;
              flip' f = λx y → f y x}
```

Even though that's the same as writing $flip'\ f\ x\ y = f\ y\ x$, we make it obvious that *flip'* will be used to produce a new function.

```
ghci 52> take 4 ['a'..'z']
    "abcd"
```

```
ghci 53> (flip' take) ['a'..'z'] 4
    "abcd"
```

3.6 Solving the right triangles problem again

Recall how we solved the problem of finding right triangles with a certain perimeter.

With imperative programming, we would have solved it by nesting three loops and then testing if the current combination satisfies Pythagoras' theorem and if it has the right perimeter.

In functional programming, that pattern is achieved with mapping and filtering. We make a function that takes a value and produces some result. We map that function over a list of values and then we filter the resulting list to get the results that we want.

Thanks to Haskell's laziness, even when we map functions over a list several times and / or filter it several times, we only pass over the list once. And even then, we only look at that part of the list that is needed to compute the final result.

Here's how we can solve the right triangles problem using lambdas and *filter*.

```
ghci 54> filter (λ(a,b,c) → a + b + c ≡ 24)
           (filter (λ(a,b,c) → a ↑ 2 + b ↑ 2 ≡ c ↑ 2)
             [(a,b,c) | c ← [1 .. 10], b ← [1 .. c], a ← [1 .. b]])
    [(6,8,10)]
```

4 Function application with $

We will now introduce the $ function, also called function application. Here's its type:

```
ghci 55> :t ($)
    ($) :: (a → b) → a → b
```

And this is its definition:

(3) $f \mathbin{\$} x = f\ x$

The $ operator seems to be completely redundant since it's just function application as denoted by the blank space. But the two are not exactly the same:

- whereas normal function application (putting a space between two things) has a really high precedence, the $ function has the lowest precedence
- function application with a space is left-associative, so $f\ a\ b\ c$ is the same as $((f\ a)\ b)\ c$, but function application with $ is right-associative, so $f \mathbin{\$} g \mathbin{\$} h \mathbin{\$} x$ is the same as $f \mathbin{\$} (g \mathbin{\$} (h \mathbin{\$} x))$

This helps us in two ways. First, $ is a convenience function that enables us to omit many parentheses. Consider the expression:

```
ghci 56> sum (map sqrt [1 .. 100])
    671.4629471031477
```

Because $ has such a low precedence, we can rewrite it as:

```
ghci 57> sum $ map sqrt [1 .. 100]
    671.4629471031477
```

When a $ is encountered, the expression on its right is passed as an argument to the function on its left.

4.1 More examples of parenthesis elimination with $

Now consider the expression below:

```
ghci 58> sqrt 3 + 4 + 9
    14.732050807568877
```

This adds together the square root of 3, 4 and 9. If we want get the square root of $3+4+9$, we'd have to write *sqrt* $(3+4+9)$ or we could just use \$:

```
ghci 59> sqrt $ 3 + 4 + 9
    4.0
```

We can do this because \$ has the lowest precedence of any operator. That's why we can imagine a \$ being equivalent to writing an opening parenthesis instead and then writing a closing one on the far right side of the expression following \$.

And here's a second example showing how \$ improves readability. We can rewrite this:

```
ghci 60> sum (filter (>10) (map (*2) [2..10]))
    80
```

as this:

```
ghci 61> sum $ filter (>10) $ map (*2) [2..10]
    80
```

4.2 Treating function application as a regular function: mapping over \$

But apart from getting rid of parentheses, \$ means that function application can be treated just like another function.

For example, we can map function application over a list of functions:

```
ghci 62> map ($3) [(4+), (10*), (↑2), sqrt, succ, pred]
    [7.0, 30.0, 9.0, 1.7320508075688772, 4.0, 2.0]
```

5 Function composition

Function composition $\circ$ is defined as follows:

(4) $(f \circ g)\ (x) = f\ (g\ (x))$

That is, composing two functions f and g produces a new function $f \circ g$. When $f \circ g$ is applied to an argument x, the result is the same as first applying g to x and then applying f to the result $g\ (x)$.

In Haskell, function composition is pretty much the same thing. We do function composition with the $\circ$ function. Its type is as follows:

```
ghci 63> :t (∘)
    (∘) :: (b → c) → (a → b) → a → c
```

And this is its definition:

(5) $f \circ g = \lambda x \rightarrow f\ (g\ x)$

As the type declaration indicates, f must take as its argument a value that has the same type as g's return value. So the resulting function takes an argument of the same type as g's argument and returns a value of the same type as f's return value.

For example, the expression $negate \circ (*3)$ returns a function that takes a number, multiplies it by 3 and then negates it.

```
ghci 64> (negate ∘ (*3)) 2
    −6
```

```
ghci 65> (negate ∘ (*3)) (−2)
    6
```

5.1 Generating functions on the fly with function composition

One of the uses for function composition is making functions on the fly to pass to other functions. We can use lambdas for that, but function composition is often 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 66> map (λx → negate (abs x)) [5, −3, −6, 7, −3, 2, −19, 24]
    [−5, −3, −6, −7, −3, −2, −19, −24]
```

Notice the lambda and how the expression in its scope looks like the definition of function composition. So using function composition, we can rewrite this as:

```
ghci 67> map (negate ∘ abs) [5, −3, −6, 7, −3, 2, −19, 24]
    [−5, −3, −6, −7, −3, −2, −19, −24]
```

5.2 Using the right-associativity of function composition

Function composition is right-associative: the expression $f\ (g\ (h\ x))$ is equivalent to $(f \circ g \circ h)\ x$.

We can use this property to compose many functions at a time. For example, we can rewrite the expression below:

```
ghci 68> map (λxs → negate (sum (tail xs))) [[1 .. 5], [3 .. 6], [1 .. 7]]
    [−14, −15, −27]
```

as the following, much more readable one:

```
ghci 69> map (negate ∘ sum ∘ tail) [[1 .. 5], [3 .. 6], [1 .. 7]]
    [−14, −15, −27]
```

5.3 Function composition and multi-argument functions

But what about functions that take several arguments?

If we want to use them in function composition, we have to partially apply them enough so that each function takes just one argument.

For example, this:

```
ghci 70> sum (replicate 5 (max 6.7 8.9))
    44.5
```

can be rewritten as this:

```
ghci 71> (sum ∘ replicate 5 ∘ max 6.7) 8.9
    44.5
```

or as this:

```
ghci 72> sum ∘ replicate 5 ∘ max 6.7 $ 8.9
    44.5
```

Although it is probably the most readable when written like this:

```
ghci 73> sum ∘ replicate 5 ∘ maximum $ [6.7, 8.9]
    44.5
```

5.4 General rules for rewriting expressions using function composition

If we want to rewrite an expression with lots of parentheses by using function composition:

- we start by putting the last parameter of the innermost function after a $;
- then we compose all the other function calls, i.e., we write them without their last parameter and we put dots (∘) between them.

For example, if we have:

```
ghci 74> replicate 10 (product (map (*3) (zipWith max [1, 2, 3, 4, 5] [4, 5, 6, 7, 8])))
    [1632960, 1632960, 1632960, 1632960, 1632960, 1632960, 1632960, 1632960, 1632960, 1632960]
```

we can write it as:

```
ghci 75> replicate 10 ∘ product ∘ map (*3) ∘ zipWith max [1, 2, 3, 4, 5] $ [4, 5, 6, 7, 8]
    [1632960, 1632960, 1632960, 1632960, 1632960, 1632960, 1632960, 1632960, 1632960, 1632960]
```

If the expression ends with three parentheses, chances are that it'll have three composition operators when it is rewritten in this style.

6 Point free style

Another common use of function composition is defining functions in the so-called point free style, i.e., without explicitly passing the argument (the 'point').

For example, how would we write the function below in point free style?

```
ghci 76> let fn x = ceiling (negate (tan (cos (max pi x))))
ghci 77>  :t fn
    fn :: (Floating a, Integral b, RealFrac a) ⇒ a → b
```

```
ghci 78> fn 0
    2
```

```
ghci 79> fn $ 2 ∗ pi
    −1
```

We can't just get rid of the x on both sides. The x in the function body (i.e., on the right side) has parentheses after it: *cos* (*max pi*) 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.

```
ghci 80> let fn = ceiling ∘ negate ∘ tan ∘ cos ∘ max pi
```

```
ghci 81>  :t fn
    fn :: Double → Integer
```

```
ghci 82> fn 0
    2
```

```
ghci 83> fn $ 2 ∗ pi
    −1
```

Many times, point free style is more readable and concise because it makes us think about functions and the result of their composition instead of thinking about data and how it's shuffled around.

Point free style really highlights that we can take simple functions and use composition as a glue to form more complex functions.

However, writing a function in point free style can be less readable if the function is too complex. That's why making long chains of function compositions is not advisable. Instead, we should use **let** or **where** bindings to give labels to intermediate functions or split the function into sub-functions and then put the sub-functions back together. This way, the overall structure of the main function is clearly represented.