New Algorithms - A Functional Perspective

This lesson gives an overview of the new algorithms that are a part of C++17.

All new functions have a pendant in the purely functional language Haskell.

Functions	Haskell
`std::for_each_n`	`map`
`std::exclusive_scan`	`scanl`
`std::inclusive_scan`	`scanl1`
`std::transform_exclusive_scan` and `std::transform_inclusive_scan`	composition of `map` and `scanl` or `scanl1`
`std::reduce`	`foldl` or `foldl1`
`transform_reduce`	composition of `map` and `foldl` or `foldl1`

Before I show you Haskell in action, let me briefly discuss the different functions.

Functions	Description
`map`	applies a function to a list
`foldl` and `foldl1`	apply a binary operation to a list and reduce the list to a value.

	and reduce the list to a value. `foldl` needs, in contrast to `foldl1`, an initial value.
`scanl` and `scanl1`	apply the same strategy as `foldl` and `foldl1` but produce all intermediate results so that you will get back a list

- Note: `foldl`, `foldl1`, `scanl`, and `scanl1` start their job from the left.

Let's have a look at the running example of Haskell functions:

```haskell
main = do let ints = [1..9]
          let strings =["Only","for","testing","purpose"]
          print (map (\a -> a * a) ints)
          print (scanl (*) 1 ints)
          print (scanl (+) 0 ints)
          print (scanl (+) 0 . map(\a -> a * a) $ints)
          print (scanl1 (+) . map(\a -> length a) $strings)
          print (foldl1 (\l r -> l++ ":" ++r) strings)
          print (foldl (+) 0 . map (\a -> length a) $strings)
```

(1) and (2) define a list of integers and a list of strings. In (3), I apply the lambda function `(\a -> a * a)` to the list of integers. That being said, (4) and (5) are more sophisticated. The expression (4) multiplies (*) all pairs of integers starting with 1 as a neutral element of multiplication. Expression (5), on the other hand, does the corresponding for addition. Expressions (6), (7), and (9) are, for the imperative eye, quite challenging; you have to read them from right to left. `scanl1 (+) . map(\a -> length)` (7) is a function composition. The dot (.) symbol composes the two functions. The first function maps each element to its length; the second function adds the list of lengths together. (9) is similar to (7), the difference being that `foldl` produces one value and requires an initial element that is 0 in this case. Now expression (8) should be readable; it successively joins two strings with the ":" character.