That new function is called the **forward composition** of the two functions **f** and **g**. In **Scala**, this operation is written as **f andThen g** …

…

The **forward composition** is denoted by $\fatsemi$ (pronounced "**before**") and can be defined as

$$f \fatsemi g \triangleq (x \Rightarrow g(f(x)))$$

The symbol $\triangleq$, means "is defined as".

We could write the **forward composition** as a fully parametric function,

```
def andThen[X, Y, Z](f: X => Y)(g: Y => Z): X => Z = { x => g(f(x)) }
```

The type signature of this curried function is

$$\text{andThen} : (X \Rightarrow Y) \Rightarrow (Y \Rightarrow Z) \Rightarrow X \Rightarrow Z$$

name: **Forward Composition**

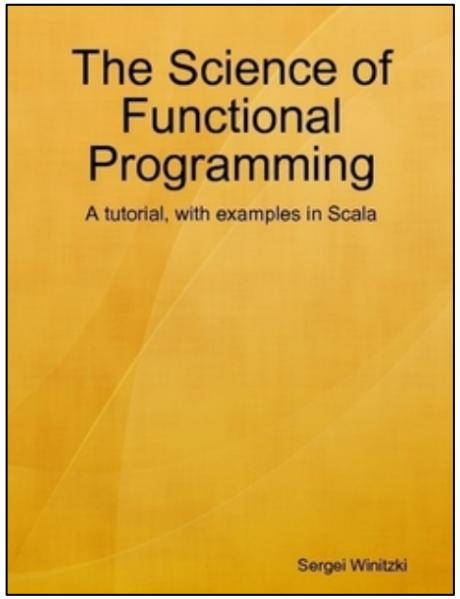
$f \fatsemi g$

pronunciation: f **before** g
in ***Scala***: f **andThen** g

name: **Backward Composition**

$f \circ g$

pronunciation: f **after** g
in ***Scala***: f **compose** g



This type signature requires the types of the function arguments to match in a certain way, or else the **composition** is undefined.

The **backward composition** of two functions **f** and **g** works in the **opposite order**: first **g** is applied and then **f** is applied to the result. Using the symbol $\circ$ (pronounced "**after**") for this operation, we can write

$$f \circ g \triangleq (x \Rightarrow f(g(x)))$$

In **Scala**, the **backward composition** is called **compose** and used as **f compose g**. This method may be Implemented as a fully parametric function

```
def compose[X, Y, Z](f: Y => X)(g: Z => Y): Z => X = { z => f(g(z)) }
```

The type signature of this curried function is

$$\text{compose} : (Y \Rightarrow X) \Rightarrow (Z \Rightarrow Y) \Rightarrow Z \Rightarrow X$$

Sergei Winitzki

sergei-winitzki-11a6431