# Chapter 1: Values, types, expressions, functions

Sergei Winitzki

Academy by the Bay

October 21, 2017

# What is "Functional Programming"?

Functional programming...

- treats programs as mathematical expressions
- uses age-old mathematical intuition to design software
- is natural and effective in OCaml, Haskell, F#, Scala, Swift, etc.
- ... but not in C, C++, JavaScript, Java (before version 8), or Python!

We will be using Scala for all examples...

- ...but the same material looks very similar in the other languages

# Examples of functional programs

Compute the factorial of a natural number:

$$n! = \prod_{k=1}^{n} k$$

Check whether a natural number is a prime:

$$prime(n) = \forall i \text{ such that } 2 \leq i < n \ : \ n \neq 0 \mod i$$

Count how many even numbers there are in a given set $S$ of integers:

$$count\_even = \sum_{k \in S} is\_even(k)$$

$$\text{where we defined } is\_even(k) = \begin{cases} 1 & \text{if } k = 0 \mod 2 \\ 0 & \text{otherwise} \end{cases}$$

- Scala programs implementing this are similar to the math
  - Programs in Java or Python are *not* similar to the math!

# What exactly is "math-like" about that code?

- The code represents a *mathematical expression* that we want to compute
- Each value is immutable and has a fixed *type* (integer, set, function, etc.)
- The code can define new names or new functions *within an expression*
- There are no loops, no "goto" or "repeat"
  - Have you ever seen a math book that says things such as,
    *"now change $k$ to $k-1$ and repeat Equation 123 until $k=0$" ?*
    or
    *"at this point, if $x>0$ then go back to page 208, else assign $x=0$"?*

# Adapting math to programming I

## Values, expressions, and types

In math, there are two kinds of "variables":

- named constant values,

```
val a = 123
```

- function arguments,

```
def f(x: Int, y: Int) = x + y - 1
```

Math texts never try to "modify" a value, so...

- "`val`"s and function arguments are immutable

Each value has a fixed type (`Int`, `Boolean`, `Set[Int]`, etc.)

- Type represents the set of possible values of the function argument
- Type is automatically assigned to named constants

# Adapting math to programming II

## Anonymous functions vs. named functions

There are two ways of defining a function in Scala:

- **named** function – using `def` with a name

```
def merge(x: Int, y: Int): Int = { x + y - 1 }
```

- **anonymous** function – in math notation, $x \mapsto f(x)$:

```
(x: Int, y: Int) => x + y - 1
```

Anonymous functions are *values*:

- they are immutable and have a fixed type, e.g. `(Int, Int) => Int`
- they can be assigned a name and used later in an expression:

```
val double: (Int => Int) = { x => x * 2 }; double(y)
```

- or they can be used directly as arguments of other functions:

```
(1 to 100).map(x => x * x)
```

# Some collections in Scala

What is (1 to 100)? What type does it have?

```
scala> (1 to 100)
res0: scala.collection.immutable.Range.Inclusive = Range 1 to 100
```

- Sequence, `Seq` and its subtypes: `List`, `IndexedSeq`, etc.

```
val a: Seq[Int] = Seq(2, 4, 6, 8)
val b = a(0) // now b: Int == 2
```

- Set: `Set`

```
val b: Set[String] = Set("x", "y", "z")
```

- Dictionary: `Map`

```
val c: Map[String, Int] = Map("x" -> 5, "y" -> 3, "z" -> 1)
```

Note the **parameterized** types `Seq[Int]` and `Map[String, Int]`
Collections can have any element type: `String`, `Boolean`, `Set[Seq[Int]]`, ...

# Adapting math to programming III

## Encoding sums, products, quantifiers using anonymous functions

The methods map, filter, forall, exists are defined on all collections
The methods sum, product are defined on collections of *numbers*

| *Mathematical notation* | *Scala code* |
|---|---|
| $x \mapsto \sqrt{x^2+1}$ | `x => math.sqrt(x * x + 1)` |
| sequence $[1,\ 2,\ ...,\ n]$ | `(1 to n)` |
| sequence $[f(1),\ ...,\ f(n)]$ | `(1 to n).map(k => f(k))` |
| $\sum_{k=1}^{n} k^2$ | `(1 to n).map(k => k*k).sum` |
| $\prod_{k=1}^{n} f(k)$ | `(1 to n).map(f).product` |
| $\forall k$ such that $1 \leq k \leq n : p(k)$ holds | `(1 to n).forall(k => p(k))` |
| $\exists k,\ 1 \leq k \leq n$ such that $p(k)$ holds | `(1 to n).exists(k => p(k))` |
| $\sum_{k \in S:\ p(k) \text{ holds}} f(k)$ | `s.filter(p).map(f).sum` |

# Adapting math to programming IV

## Higher-order functions

Derivatives and integrals could be implemented as functions:

```
def deriv(f: Double => Double): (Double => Double) = { ??? }
def integ(f: Double => Double, range: (Double, Double)): Double = ???
```

**Higher-order functions** take function arguments and/or return functions

- Many computations with sequences, sets, dictionaries can be done using higher-order functions, *without loops*, concisely and error-free
- The Scala standard library has many more higher-order methods for collections
  - `size`, `reduce`, `zip`, `zipWithIndex`, `flatten`, `flatMap`, `foldLeft`, `foldRight`, `scanLeft`, `collect`, `distinct`, `groupBy`, ...
- Write code by formulating the problem as a mathematical expression

# Examples

- Using both `def` and `val`, define a function that...
  1. adds 20 to its integer argument
  2. takes an integer `x`, and returns a function that adds `x` to its argument
  3. takes an integer `x`, and returns `true` iff `x+1` is prime
- What are the types of the functions in Examples 1 - 3?
- Compute the average of all numbers in a sequence of type `Seq[Double]`. Use `sum` and `size` but no loops.
- Given $n$, compute the Wallis product truncated up to $\frac{2n}{2n+1}$:

$$\frac{2}{1}\frac{2}{3}\frac{4}{3}\frac{4}{5}\frac{6}{5}\frac{6}{7}\cdots\frac{2n}{2n+1}.$$

  Use a sequence of `Int` or `Double` numbers, `map`, and `product`.
- Given `s: Seq[Set[Int]]`, compute the sequence containing the sets of size at least 3. Use `map`, `filter`, `size`. The result must be again of type `Seq[Set[Int]]`.

# Summary

What problems can we solve now?

- Compute mathematical expressions involving sums, products, and quantifiers, based on integer ranges (such as $\sum_{k=1}^{n} f(k)$ etc.)
- Implement functions that take or return other functions
- Work on collections using `map` and other library methods

What kinds of problems are not solved with these tools?

- Compute the smallest $n$ such that $f(f(f(...f(1)...) > 1000$, where the function $f()$ is applied $n$ times.
- Find the median in an array of integers.

Why can't we solve such problems yet?

- Because we can't yet put *mathematical induction* into code

# Exercises

1. Define a function of type `Seq[Double] => Seq[Double]` that "normalizes" the sequence: it finds the element having the max. absolute value and, if that value is nonzero, divides all elements by that factor.
2. Define a function of type `Seq[Seq[Int]] => Seq[Seq[Int]]` that adds 20 to every element of every inner sequence.
3. An integer $n$ is called "two-factor" if it is divisible by only two different integers $j$ such that $2 \leq j < n$. Compute the set of all "two-factor" integers $n$ among $n \in [1, ..., 100]$ .
4. Given a function $f$ of type `Int => Boolean`, an integer $n$ is called "two-$f$" if there are only two different integers $j \in [1, ..., n]$ such that $f(j)$ returns `true`. Define a function that takes $f$ as an argument and returns a sequence of all "two-$f$" integers among $n \in [1, ..., 100]$. What is the type of that function? Rewrite Exercise 3 using that function.
5. Define a function that takes two functions `f: Int => Double` and `g: Double => String` as arguments, and returns a new function that computes the functional composition of `f` and `g`.