Programming paradigms:

* imperative programming
* functional programming
* logic programming

A theory consists of:

* one or more data types
* operations on these data types
* laws that describe the relationships between values and operations

Normally, a theory doesn’t describe mutations.

For instance the theory of polynomials defines the sum of two polynomials by laws:

(a\*x + b) + (c\*x + d) = (a+c)\*x + (b + d)

A theory **doesn’t define** an operator to change a coefficient while keeping the polynomial the same.

Another example:

The theory of strings defines a concatenation operator which is associative:

(a ++ b) ++ c = a ++ (b ++ c)

But it **doesn’t define** an operator to change a sequence element while keeping the sequence the same.

In a restricted sense, FP means programming without mutable variables, assignments, loops, and other imperative control structures.

In a wider sense, FP means focusing on the functions. Functions can be produced, consumed, and composed.

**Call-by-value** evaluation strategy is the evaluation strategy that evaluates every function argument only once because it evaluates arguments values first. **Call-by-name** evaluates function parameters only after evaluation of the function body. It has an advantage that a function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body.

Both strategies **reduce to the same result** as long as the reduced expression consists of pure functions and both evaluations terminate.

* if CBV evaluation of an expression e terminates then **CBN evaluation of e terminates as well**
* the other direction is not true

Example of termination under CBN but not termination under CBV is:

def first(x: Int, y: Int) = x

And let’s consider the expression first(1, loop).  
Scala normally uses call-by-value, but **if the type of a function parameter starts with => it uses call-by-name.**

&& and || doesn’t always need the right side of the expression (like for the case true || e) in order for it to be evaluated. These expressions use **‘short-circuit evaluation’.**

We have seen that function parameters can be passed by value or can be passed by name. The same distinction applies to definitions. The def form is ‘by-name’, **its right-hand side is evaluated on each use.** val form is ‘by-value’.

For example,

val y = square(x)

The right-hand side of a val definition is evaluated at the point of the definition itself. Afterwards, the name refers to the value.

For example, given

def loop: Boolean = loop

A definition

def x = loop

Is ok, but a definition

val x = loop

Will lead to an infinite loop.

We can also calculate the square root of the value using Newton’s method. In order to calculate the square root of x sqrt(x) we start with the initial estimate y (let’s pick y = 1) and then repeatedly improve the estimate by taking the mean of y and x/y.

Example (for x = 2) :

Estimation Quotient Mean

1 2 / 1 = 2 1.5

1.5 2 / 1.5 = 1.333 1.4167

1.4167 2 / 1.4167 = 1.4118 1.4142

1.4142 … …

First, let’s define a function that computes one iteration step.

def sqrtIter(guess: Double, x: Double): Double =

if (isGoodEnough(guess, x)) guess

else sqrtIter(improve(guess, x), x)

Recursive functions **need an explicit return type in Scala** (the function is recursive if its right-hand side calls itself). For non-recursive functions, the return type is optional.

We can define isGoodEnough function this way:

def isGoodEnough(guess: Double, x: Double) =

abs(guess \* guess - x) / x < 0.01

The core of the Newton method is improve function:

def improve(guess: Double, x: Double) =

((x / guess) + guess) / 2

The square root method itself would be:

def sqrt(x: Double) = sqrtIter(1.0, x)

In Scala we can also use blocks, which are delimited by braces. A block contains a sequence of definitions or expressions. The last element of a block is an expression which defines its value. Blocks are themselves expressions.

val x = 0

def f(y: Int) = y + 1

val result = {

val x = f(3)

x \* x

}

The definitions inside the block are only visible within the block. The definitions inside the block shadow the definitions of the same names outside the block.

Multi-line expressions should be written in parentheses or the operator should be written in the first line because this tells the Scala compiler that the expression is not yet finished.

someLongExpression +

someOtherLongExpression

If a function calls itself as its last action, the function’s stack frame can be reused. This is called **tail recursion.** Tail recursive functions are iterative processes.

In general, if the last call of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions. Such calls are called **tail-calls.**

gcd, the function that computes the greatest common divisor of two numbers can be implemented using Euclid’s algorithm:

def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

Consider factorial:

def factorial(n: Int): Int =

if (n == 0) 1 else n \* factorial(n - 1)

gcd function is a tail recursive function, factorial is not a tail recursive function.

In Scala, only directly recursive calls to the current function are optimized. One can require a function to be tail recursive using @tailrec annotation. If the annotation is given, and the function is not tail recursive, an error would be issued.

Tail-recursive version of factorial is:

def factorial(n: Int): Int = {

def loop(acc: Int, n: Int): Int =

if (n == 0) acc

else loop(acc \* n, n - 1)

loop(1, n)

}