Elements of Programming

Every non-trivial programming language provides:

- primitive expressions representing the simplest elements
- ways to *combine* expressions
- ways to *abstract* expressions, which introduce a name for an expression by which it can then be referred to.

So in this week, we are going to do some Scala programming. We're going to take the first steps into this new programming language and into functional programming in general. A lot of the material in this session will look very easy to you because it's familiar. But there are also some things that are fundamental, in particular, the former model of evaluation that we call the substitution model, which will be very important for the sessions later on. So, it's good to pay a little attention. Okay, let's get into it. Every non-trivial language provides primitive expressions that represents the simplest elements of the language. And then some ways to combine expressions and ways to abstract expressions. Abstraction means that we introduce a name for an expression and then further on we can reference the expression by its name.

One way to approach functional programming is to see it a little bit like a calculator. In fact most functional languages have an interactive shell. That's also sometimes called a REPL, which stands for read, eval, print, loop, and in that shell you can type expressions, and it will respond with the results of evaluating these expressions. In Scala you can start the REPL simply by typing “scala” (在我電腦上可以).

The Read-Eval-Print Loop

Functional programming is a bit like using a calculator

An interactive shell (or REPL, for Read-Eval-Print-Loop) lets one write expressions and responds with their value.

The Scala REPL can be started by simply typing

```
> scala
```

```
scala> 34+65
res0: Int = 99

scala> def radius = 10
radius: Int

scala> def pi = 3.14159
pi: Double

scala> radius * pi
res1: Double = 31.4159
```

So if you write Scala here, Then you get the ripple, that assumes that you have a standard Scala distribution. That's actually not a requirement for this course. For the course, we ask you to install SBT, the Scala build tool instead. From the SBT, you can also get the ripple simply by typing "sbt consol". It's the same REPL, so it doesn't matter whether you start that with Scala and SBT consol. Once you're on the REPL, you can type an expression like 34+65. You can also define values. Such as radius = 10. And REPL, in each case, will respond, if you have an expression, with the expression's type, in this case that is Int, and the expression's value. If you do have a definition like radius, it would just give you the type of the definition. You can also define floating point numbers. And afterwards you can refer To what you have defined before.

Evaluation

A non-primitive expression is evaluated as follows.

1. Take the leftmost operator
2. Evaluate its operands (left before right)
3. Apply the operator to the operands

A name is evaluated by replacing it with the right hand side of its definition

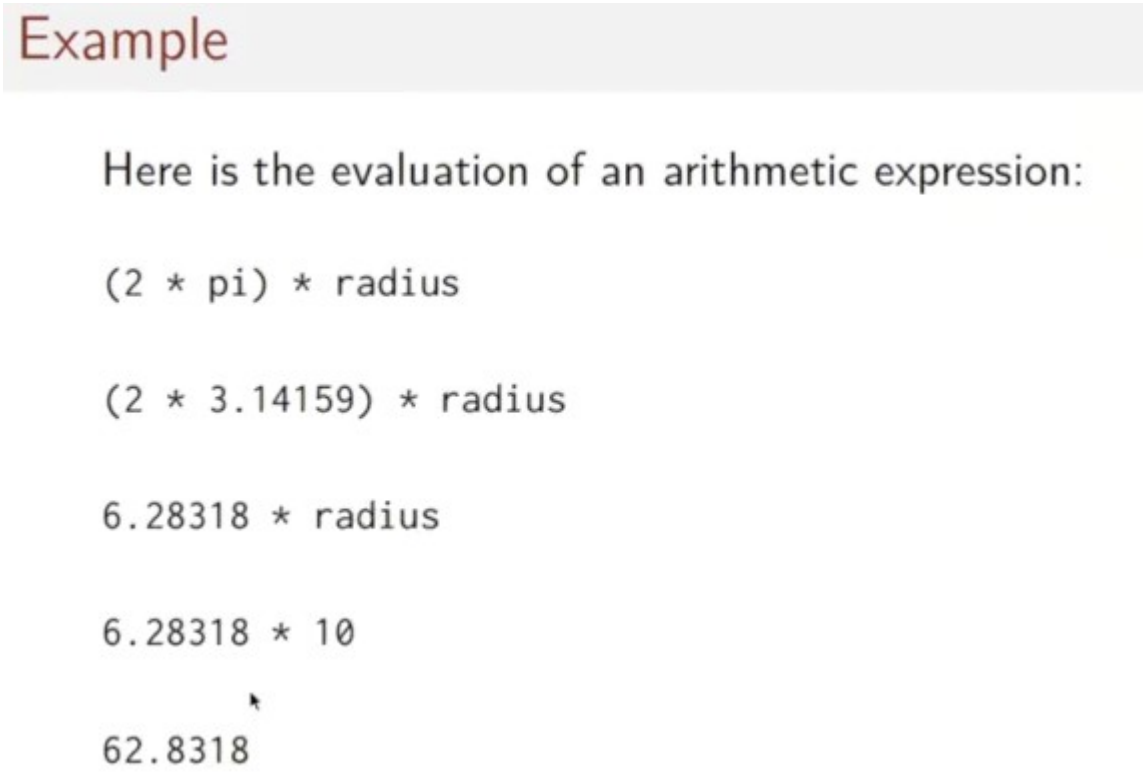
The evaluation process stops once it results in a value

A value is a number (for the moment)

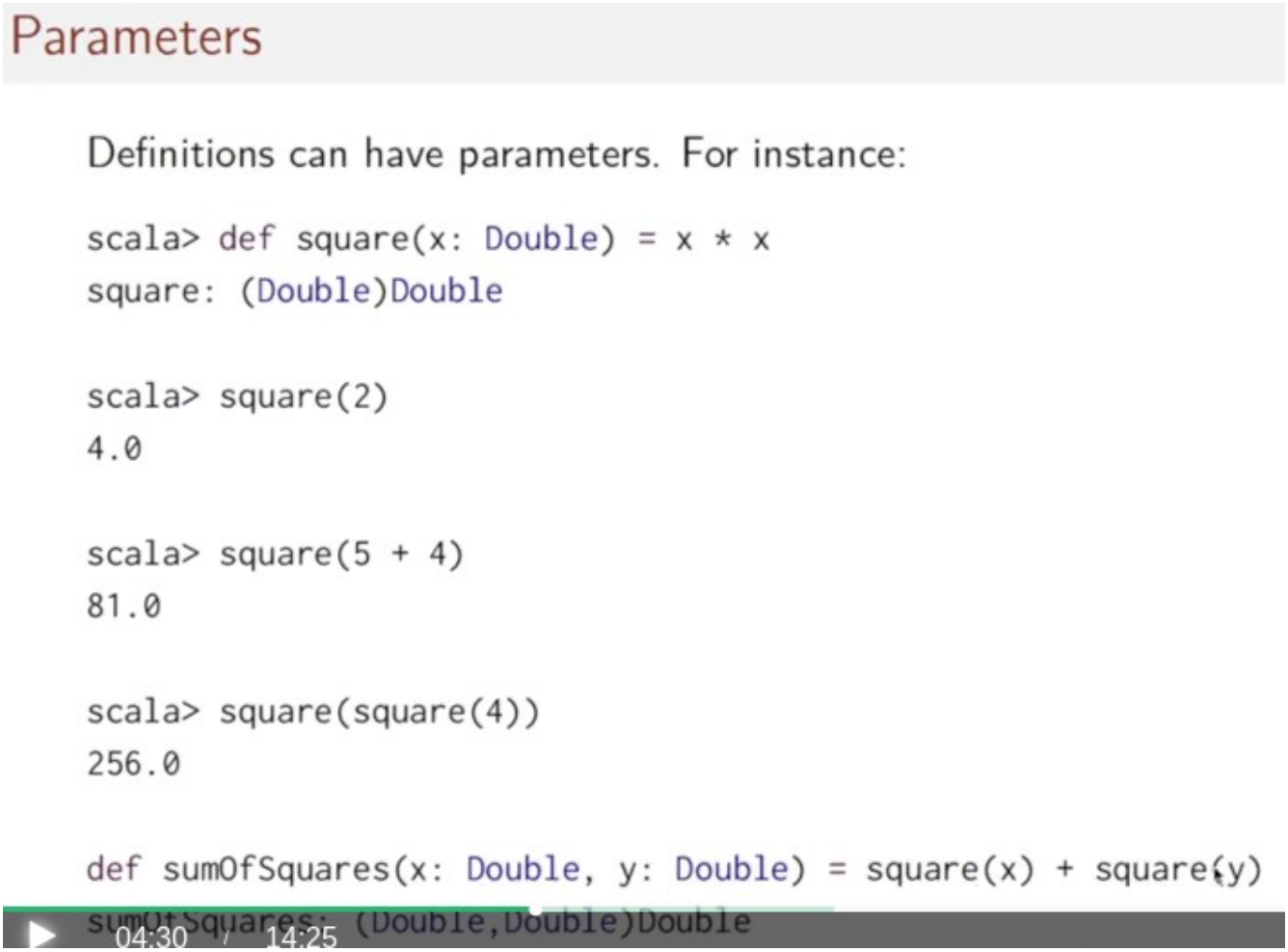
Later on we will consider also other kinds of values

Now, let's have a look at evaluation. You have seen how that these expressions were evaluated. How exactly did that happen? Well. The rules are well known from simple algebra. To evaluate a non-primitive expression, we typically take the left most operator, subject to the rules of precedence. We evaluate the operands of that operator. Typically left before right. And we apply the operator to the operands. What about evaluating a name? Well, that's evaluated by replacing it with the right hand side

of its definition. We applied these steps one by one, until finally, the evaluation process results in a value. And for the moment the value is just a number, later on, we also consider other kinds of values.



So let's see an example, here's the evaluation of the arithmetic expression, two times pi times radius. What's happening there, first thing we do is we look up the name pi and we get its definition, 3.14159. Then we perform the arithmetic operation on the left. Then we look up the radius evaluate radius yielding ten, and finally, we perform the final multiplication, yielding the result.



Definitions can also have parameters. For instance, we can define a square function by writing def square. Then comes the parameter x of type double, and then comes the right hand side, x times x. The right hand side refers to the parameter x. And the p evaluation then would yield the expected square of two, would give four, a square of five times four would give 81, We can, of course, also have, parameterized definitions that refer to parameterized functions as in the sum of squares function here, that takes two parameters, x and y of type double, and it computes the square x and square y and sums them.

Parameter and Return Types

Function parameters come with their type, which is given after a colon

```
def power(x: Double, y: Int): Double = ...
```

If a return type is given, it follows the parameter list.

Primitive types are as in Java, but are written capitalized, e.g:

`Int`	32-bit integers
`Double`	64-bit floating point numbers
`Boolean`	boolean values `true` and `false`

We've seen in the last slide that function parameters come with a type, and this type is given after the parameter name and a colon in Scala. So you would write x colon double y colon int. You can also give the return type of a function that follows after the parameter list. The permative types are written as in Java, but they are capitalized. So, int for example, is a 32-bit integer, double is a 64-bit, 40-point number, and Boolean represents a Boolean type bit value true and false.

Evaluation of Function Applications

Applications of parameterized functions are evaluated in a similar way as operators:

1. Evaluate all function arguments, from left to right
2. Replace the function application by the function's right-hand side, and, at the same time
3. Replace the formal parameters of the function by the actual arguments.

Now, how are function applications evaluated? In fact, they are evaluated in a way that is very similar to operators. We first evaluate all function arguments going from left to right. We then replace the function application by the function's right hand side, and at the same time, we replace the former parameters of the function by the actual arguments.

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
25
```

Let's see how this works in an example. We start with sum of squares of three and two + two. The two + two gets rewritten to a value, four, and we and, we have sum of squares three, four. Then we take the definition of sum of squares, that was square of x plus square of y, and we replace the application by this right hand side, where at the same time we replace the parameters x and y by the actual arguments three and four. We then repeat the process with the square applications. The square of three becomes three times three. The right turn side of square where three replaces the promidence. We get an arithmetic simplification, and we do the same thing for the right hand operators yielding the final value 25.

The substitution model

This scheme of expression evaluation is called the *substitution model*.

The idea underlying this model is that all evaluation does is *reduce an expression to a value*.

It can be applied to all expressions, as long as they have no side effects.

The substitution model is formalized in the *λ-calculus*, which gives a foundation for functional programming.

C++

This scheme of expression evaluation is called the substitution model. The idea underlying this model is that all evaluation does is reduce an expression to a value. And that reduction can be expressed by a sequence of simple rewriting steps that rewrite the expression term itself until it is a value, very similar to what you would do in algebraic simplification. Simple as it is, this model is very powerful. In fact it has been shown that it can express every algorithm, so it's equivalent to what you would call a Turing machine. This has been shown a long time before functional programming, in fact, a long time before computers by a logician called Alonzo Church in the lambda calculus. He did that in the thirties of the last century. You might have come across lambda calculus in theoretical computer science, and if you did, then you know how to make the connection to this model. If you didn't, don't worry. We won't need the theory for following the course. What's important though is to know that number calculus as a model, and the substitution model, can be applied only to expressions that do not have a side effect. Now, what is a side effect? A typical side effect would be an expression called. C++ where c is a variable. And evaluating this expression means that you return the old value of c, and at the same time you increment the value. So, at the next time you return the value, you would return the value one larger than before. Turns out there's no good way to represent this evaluation sequence by a simple rewriting of this term. You need something else, like store, where the current value of the variable is kept. In other words the expression C++ has a side effect on the current value of the variable. And that side effect cannot be expressed by the substitution model. So one of the motivations for ruling out side effects in function programming is that we can keep to the simple model of evaluation.

Termination

- *Does every expression reduce to a value (in a finite number of steps)?*
- *No. Here is a counter-example*

```
def loop: Int = loop
```

loop → loop → . . .

Once we have the substitution model, another question comes up. Does every expression reduce to a value in a finite number of steps? In fact, the answer is no. Here's a counter example. Let's write simple function def loop int equals loop. And that's then called, referred to the name loop. What would happen. Well, according to our model we have to de evaluate that name which happens by replacing the name by its right hand side. The right hand side is looped again. So we have reduced the name to itself

and this can go on ad infinitum. Another way to visualize this reduction sequence is by starting with the loop and then reducing to the same term again.

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
9 + 4 * 4
25
```

We've seen that the Scala interpreter reduces function arguments to values before rewriting the function application. That's not the only possible reduction strategy. Alternatively, one could apply the function to unreduced arguments. For instance, we could start with some of squares three, two plus two, and then go on as follows. We keep the right hand side. We don't reduce it to four and we simply pass it as an expression to the square function. We then simplify the right hand side as before. And then do the same thing again, pass the expression, two plus two to this occurrence of square, that gives two plus two twice in the multiplication. We then simplify the two further steps to arrive at the final result, 25.

Call-by-name and call-by-value

The first evaluation strategy is known as *call-by-value*, the second is is known as *call-by-name*.

Both strategies reduce to the same final values as long as

- the reduced expression consists of pure functions, and
- both evaluations terminate.

Call-by-value has the advantage that it evaluates every function argument only once.

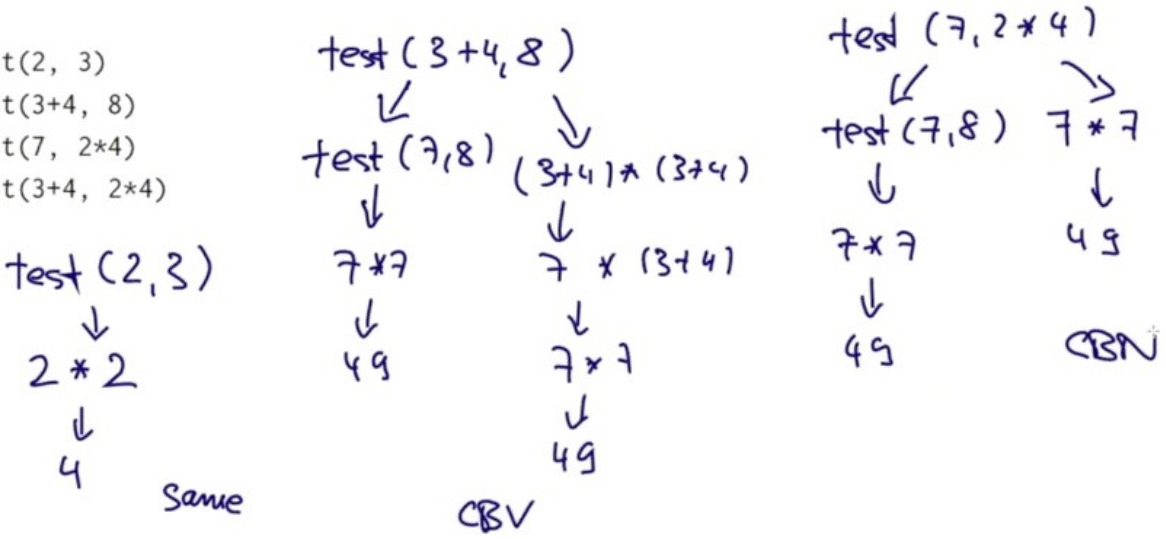
Call-by-name has the advantage that a function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body.

Now you've seen two ways to evaluate the same expression. The first evaluation strategy is known by call by value. The second is known as call by name. An important theorem, in fact, of lambda calculus is that both strategies reduce to the same final value as long as the reduced expression consists of pure functions and both evaluations terminate. Call-by-value has the advantage that every function argument is evaluated only once. Call-by-name has the advantage that a function argument is not evaluated at all if the corresponding parameter is not used in the evaluation of the function's body.

Call-by-name vs call-by-value

```
def test(x: Int, y: Int) = x * x

test(2, 3)
test(3+4, 8)
test(7, 2*4)
test(3+4, 2*4)
```



Here's a question, say you're given the following function definition: Def test of x and y equals x times x. For each of the following function applications, indicate which evaluation strategy is fastest, by which we mean, has the fewest reduction steps. That's for test of two three then test of three plus four eight. Test seven two times four, then finally, test three plus four, two times four. Let's see how we would answer this question. I have the test function here and I have the calls that I want to compare here. Let's start with the first one, test two, three. How would that evaluate? Well, test two three, we take the definition of test, x times x. So, that would give two times two and that would reduce to four. And in fact that would reduce to four on their both evaluation strategies, both core by name and core by value because we have started with already evaluated arguments so there's no choice in the matter. So here the answer would be, they are, they have the same complexity. Let's do the next one. Test three plus four eight. And I call by value, we have to evaluate the arguments. So we get test. 7,8. And that gives us seven times seven. And the final result. Where it's under called by name we get three plus four times three plus four. That reduces to seven times three plus four. Seven times seven and 49. In one step more than in the core by value version. So core by value, has the advantage here. Let's look at the third example. Test. Of seven and two times four. Here with call by value, we would get test of seven and eight. Finally, seven, seven. And the final, whereas with call by name, we would get seven. Times 749. So in this case, we have avoided the unnecessary computation for the second argument and the call by name is faster. The fourth example, test three plus four two, two times four combines the evaluation of the second and third example. So it comes at no surprise that it's again a draw, that core by value and core by name, reduces the same number of steps