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.
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
Expressions

Here are some simple interactions with the REPL

```scala
scala> 87 + 145
232
```

Functional programming languages are more than simple calculators because they let one define values and functions:

```scala
scala> def size = 2
size: => Int

scala> 5 * size
10
```
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
Example

Here is the evaluation of an arithmetic expression:

\[(2 \times \pi) \times \text{radius}\]
Example

Here is the evaluation of an arithmetic expression:

\( (2 \times \text{pi}) \times \text{radius} \)

\( (2 \times 3.14159) \times \text{radius} \)
Here is the evaluation of an arithmetic expression:

(2 * pi) * radius

(2 * 3.14159) * radius

6.28318 * radius
Example

Here is the evaluation of an arithmetic expression:

(2 * pi) * radius

(2 * 3.14159) * radius

6.28318 * radius

6.28318 * 10
Example

Here is the evaluation of an arithmetic expression:

\[(2 \times \pi) \times \text{radius}\]

\[(2 \times 3.14159) \times \text{radius}\]

\[6.28318 \times \text{radius}\]

\[6.28318 \times 10\]

\[62.8318\]
Parameters

Definitions can have parameters. For instance:

```scala
scala> def square(x: Double) = x * x
square: (Double)Double

scala> square(2)
4.0

scala> square(5 + 4)
81.0

scala> square(square(4))
256.0

def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (Double,Double)Double
```
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`
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.
Example

sumOfSquares(3, 2+2)
Example

sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
Example

sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
Example

sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
Example

sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
Example

sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
Example

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

\[
\text{sumOfSquares}(3, 2+2) \\
\text{sumOfSquares}(3, 4) \\
\text{square}(3) + \text{square}(4) \\
3 \times 3 + \text{square}(4) \\
9 + \text{square}(4) \\
9 + 4 \times 4 \\
9 + 16 \\
25
\]
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 $\lambda$-calculus, which gives a foundation for functional programming.
Termination

- Does every expression reduce to a value (in a finite number of steps)?
Termination

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

```python
def loop: Int = loop

loop
```
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)
```
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:

\[
\text{sumOfSquares}(3, 2+2) \\
\text{square}(3) + \text{square}(2+2)
\]
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:

\[
\begin{align*}
\text{sumOfSquares}(3, 2+2) \\
\text{square}(3) + \text{square}(2+2) \\
3 \times 3 + \text{square}(2+2)
\end{align*}
\]
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)
```
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:

\[
\begin{align*}
\text{sumOfSquares}(3, 2+2) \\
\text{square}(3) + \text{square}(2+2) \\
3 \times 3 + \text{square}(2+2) \\
9 + \text{square}(2+2) \\
9 + (2+2) \times (2+2)
\end{align*}
\]
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)
```
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:

\[
\begin{align*}
\text{sumOfSquares}(3, 2+2) \\
square(3) + \text{square}(2+2) \\
3 \times 3 + \text{square}(2+2) \\
9 + \text{square}(2+2) \\
9 + (2+2) \times (2+2) \\
9 + 4 \times (2+2) \\
9 + 4 \times 4
\end{align*}
\]
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
```
Call-by-name and call-by-value

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

Question: Say you are given the following function definition:

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

For each of the following function applications, indicate which evaluation strategy is fastest (has the fewest reduction steps)

<table>
<thead>
<tr>
<th>CBV</th>
<th>CBN</th>
<th>same</th>
<th>#steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>test(2, 3)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>test(3+4, 8)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>test(7, 2*4)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>test(3+4, 2*4)</td>
</tr>
</tbody>
</table>
Call-by-name vs call-by-value

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

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