Scala

Week 1

Eclipse + SBT

SBT

go to example (project) dir

type sbt, wait
type console – have fun with Scala console

Ctrl+D to exit

for submitting:
submit email password

Programming Paradigms

Theory consists of

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

no mutations!

We want:

- to concentrate on theories
- to ~~avoid~~ mutation
- to have powerful ways to abstract and compose functions

Functional Programming

- In a restricted sence, FP - programming without mutable variables, loops, assignments and other imperative control structures

- In a wider sence - FP means focusing on the functions

- Functions can be produced, consumed and composed. They are first-class citizens
  - they can be defined everythere, including inside other functions
  - can be passed as parameters and returned as results
  - we can compose functions into other functions

Why FP?

- simpler reasoning principles
- better modularity
- good for parallelism

Elements of Programming

- primitive expressions representing the simplest elements
- ways to combine expressions
- ways to abstract expressions – introduces a name by which it can be referred to

def radius = 10 ← definition
def pi = 3.14159

Evaluation of expressions

- take the leftmost operator
- evaluate its operands (left before right)
- apply the operator to the operands

(2 * pi) * radius
(2 * 3.14) * radius
...

Definitions can have parameters

def square(x: Double) = x * x

square(2) → 4

def sumOfSquare(x: Double, y: Double) =
square(x) + square(y)

Parameters and Return types

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

parameter types $\qquad$ return type

Evaluation for a function

- evaluate all function arguments from left to right
- replace the function application by the function's body and
- replace the parameters by the actual arguments

This is called the substitution model

main idea: to reduce an expression to a value

- can be applied to all expressions - as long as they don't have side effects

It's called the λ-calculus - foundation for FP

Side effect - change in definitions (like C++)

cannot be expressed by the substitution model

Evaluation strategies

- call-by-value (CBV)
  evaluates arguments before
  passing them to function

$$\begin{aligned}&\text{square}(2+2)\\&\text{square}(4)\\&4*4\\&16\end{aligned}$$

+
evaluates only
once

- call-by-name (CBN)

$$\begin{aligned}&\text{square}(2+2)\\&(2+2)*(2+2)\\&4*(2+2)\\&4*4\\&16\end{aligned}$$

+
not evaluated if
the parameter is
unused thereafter

Both strategies reduce to the same result.
as long as they don't have side effects

Termination.

- if CBV terminates, CBE also terminates
- opposite is not true

eg. def loop = loop
def first(x : Int, y: Int) = x

first(1, loop)

CBN	CBV
1	first(1, loop) → inf loop
Stop	

Scala uses call-by-value
But call-by-name can be imposed
when needed

```
def constOne(x: Int, y: => Int) = 1
```

CBV CBN

Conditionals

if-else – to express choosing between 2 alternatives

```
def abs(x: Int) = if (x >= 0) x else -x
```

x >= 0

Boolean expression

yields True or False

expression, not statement

Booleans: true, false, !b, a && b, a || b

lazy lazy

"short-circuit evaluation"

Rewrite rules for the evaluation model

if (true) e_1 then e_2 => e_1

if (false) e_1 then e_2 => e_2

Definitions

Definitions can also be „called by name"
and „called by value"

def - is a „by-name" form, its right hand
side is evaluated on each use

val - is a „by-value" form

val x = 2
val y = square(2)

val z = square(x)

its right hand side is evaluated at the
point of the definition (right away)

and afterwards, the name refers to
the value

y refers to 4, not to square(2)

def x = loop - ok
val x = loop - hangs

Example

def sqrt(x: Double): Double

by approx. using Newton's method

to compute sqrt(x):

- start with an initial guess y (y = 1 eg)
- repeatedly improve the guess by taking the mean of y and x/y

```
(1) 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, For non-recursive calls, it's optional

```
def isGoodEnough(guess: Double, x: Double) =
    abs(guess * guess - x) < 0.001

def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

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

problem with isGoodThough _
too big eps for small and
too small eps for big

$\Downarrow$ let's make it proportional to x

```
def isGoodEnough(guess: Double, x: Double) =
    abs(guess * guess - x) / x < 0.001
```

Blocks and Lexical Scoping

it's a good style to split up a task into many small functions

but sqrtIter, isGoodEnough etc matter for implementation, not for usage

Normally, we wouldn't want the user to have an access to them

=> we can put them inside SQRT! and avoid „name space pollution"

```
def sqrt(... ) = .. {
    def sqrtIter(...) =
    def isGoodEnough(...)
    def improve(...)
    sqrtIter(1.0, x)
}
```

Blocks - { } inside brackets

themselves an expression!

~~definitions~~

Visibility

- definitions inside the block are not visible outside the block - only within
- the definitions from the outside are visible inside the block
- inside definitions shadow definitions of the same names outside the block

expression!

```
val x = 0
val res = {
  val x = f(3)
  x * x
} + x
```

$f = y + 1$

It's called Lexical Scoping

and we can eliminate all occurrences of x within the block of sqrt.

```
def sqrt(x: Double) = {

  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess))

  def isGoodEnough(guess: Double) =
    abs(guess * guess - x) / x < 0.001

  def improve(guess: Double) =
    (guess + x / guess) / 2

  sqrtIter(1.0)

}
```

Semicolons - in Scala are optional

```
val x = 1;
val y = 2; y + x
```

Tail Recursion

```
def gcd(a: Int, b: Int): Int =
  if (b == 0) a else gcd(b, a % b)

def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

Difference: in factorial eg. our expression gets bigger and bigger when we evaluate

[4 * (3 * (2 * (1 * 1))) → 120]

If a function calls itself as its last action, the function's stack frame can be reused

This is called <u>tail recursion</u>

=> can be expressed in a loop with constanst space

@tailrec annotation in Scala

if it cannot be tail-recursed, an error will be thrown