Functional programming

Week 1

Evaluation strategies: call by name vs call by value

Suppose we have a program in scala that runs `sumOfSquares(3, 2+2)`. The call-by-value model would produce:

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

Whereas the call-by-name model would produce

```
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
```

The advantage of call-by-value is that every function argument is evaluted only once. In the second snippet, we see how 2+2 is evaluated twice. However, call-by-name has the advantage that a function argument is not evaluated if a parameter is unused in the evaluation of the function body.

Consider this example:

```
def test(x: Int, y: Int) = x*x
// which evaluation method is more efficient?
test(3+4,8)

//CBV
7*8
//CBN
(3+4)*8
3*8 + 4*8
```

We can also have that a program will terminate under CBN but not under CBV and vice versa. For instance:

```
def first(x: Int, y: Int) = x
first(1,loop)
```

```
//CBN
1
//CBV
first(1,loop)
.
.
.
.
```

Scala normally uses CBV. But if the type of a function parameter starts with a right arrow it will use CBN. To demonstrate this, we define the and function making sure its second argument is call by name:

```
def and(x: Boolean, y: => Boolean): Boolean =
if x then y else false
```

Week 2

Higher order and anonymous functions

A higher order function is a function that takes another function as argument. Here is an example:

Suppose we want to take the sum of all integers between some lower bound a and an upper bound b. We may recursively define

```
def sumInts(x:Int, y:Int):Int =
    if (x > y) 0
    else x + sumInts(x+1,y)
```

We could also define the same but for cubes

```
def sumCubes(x:Int, y:Int):Int =
    if (x > y) 0
    else cube(x) + sumCubes(x+1,y)
```

However we could make the code more generic and define a general sum function that takes as first argument a function, namely the rule to apply and this way if we also wanted to define some `sumFactorial` we could simply use the `sum` function. Consider

```
def sum(f: Int -> Int, a:Int, b:Int):Int =
    if a > b then 0
    else f(a) + sum(f, a+1, b)
```

Then we could easily define

```
def id(x: Int):Int = x
def sumInts(a: Int, b: Int):Int = sum(id, a, b)
```

Now notice how we tediously had to define an `id` function. Instead we may define `id` anonymously, it would then look like

```
(x:Int) -> x
```

and then define `sumInts` as

```
def sumInts(a: Int, b: Int):Int = sum((x:Int) -> x, a, b)
```

Anonymous functions are *syntatic sugar*, that is they make life nicer, but not really essential since we can always go the tedious def way.

Currying

The idea behind currying(named after Haskell Curry) is that we are able to desribe a function that takes multiple arguments as a composition of functions that all take one argument. The point of currying is that it takes a function and provides a new function with the parameter applied. For instance, we apply currying to find the product of the square of numbers in a given range as follows:

```
def product(f: Int => Int)(a:Int, b:Int): Int =
    if a > b then 1 else f(a) * product(f)(a+1,b)

// function call
product(x => x*x)(1,5)
/**
will print 14400 = 4*9*16*25
*/
```

Functions and Data

Suppose we want to define a *rational* type, we would say:

```
class Rational(x:Int, y:Int): Rational
    def numer = x
    def denom = y
```

And we would add methods to our class as follows:

```
def addRational(r:Rational, s:Rational): Rational=
    Rational (
  r.numer*s.denom+s.numer*r.denom, r.denom*s.denom)
```

Week 3

Classes and polymorphism

Abstract classes contain members which are missing an implementation, no direct instances can be created.

Persistent data structures are those data structures that can be made from preexisting ones. Consider an implementation of a set as a binary tree. Now suppose we have a tree with nodes 1,2,4,5. If we were to add 3 to this set, we would simply add it to the preexisting object.

Overriding example:

```
abstract class Base:
    def foo = 1

class Sub extends Base:
    override def foo = 2
```

An object and a class can have the same name since the two live in different namespaces. But, a class and object with the same are called **companions**. This definition is similar to static class definitions.

And here is how one creates standalone Scala code without needing the REPL:

```
object HelloWorld:
    def main(args: Array[String]) : Unit = println("hello world")
```

Class imports:

```
import week3.rational
import week3.{rational,hello}
import week3._ //imports everything
```

Traits Normally a class can have only one superclass. But what if a class shoyld have more than one super type? We use a trait.

Important top types:

- Any: base type of all types
- AnyRef: base type of all reference types
- AnyVal: base type of all primitive types

Lists: Lists in scala are defined as immutable linked-lists constructed from:

- Nil == the empty list
- Cons == cell containing element and the remainder of the list

```
trait List[T]:
    def isEmpty: Boolean
    def head: T
  def tail: List[T]

class Cons[T](val head: T, val tail: List[T]) extends List[T]:
    def isEmpty = false

class Nil[T] extends List[T]:
    def isEmpty = true
    def head = throw new NoSuchElementException("Nil.head")
    def tail = throw new NoSuchElementException("Nil.tail")
```

Value parameters: Writing (val head: Int, val tail: IntList) is the same as declaring the two parameters in the body of the class.

Type erasure: Scala removes types at compile time

Polymorphism: means a function that comes in many forms for instance:

- function can be applied to arguments of many types
- type can have instances of many types

Week 4

Our objective this week is to find a general and convenient way to access heterogeneous data in a class hierarchy. We will use **pattern matching** for this feat.

Case classes

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Var(name: String) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr
// example usage

def eval(e: Expr): Int = e match
    case Number(n) => n
    case Sum(e1,e2) => eval(e1) + eval(e2)

def show(e: Expr): String = e match
    case Number(n) => n.toString
    case Sum(e1, e2) => s"${show(e1)} + ${show(e2)}"
    case Var(x) => x
    case Prod(e1,e2) => s"${showP(e1)} + ${showP(e2)}"

def showP(e: Expr): String = e match
    case e: Sum => s"(${show(e)})"
    case _ => show(e)
```

Lists

- Lists in scala are immutable, their elements can not be changed
- Lists are recursive, arrays are flat

All lists are constructed from the empty list `Nil` and `::` (cons)

Example:

```
nums = 1 :: (2 :: (3 :: (4 :: Nil)))
```

Some facts:

- Operations that end in `:` associate to the right.
- All operations on lists can be expressed in terms of `head, tail, isEmpty`

List pattern matching example:

The pattern `x :: y :: List(xs, ys) :: zs` is matched by the condition `L >= 3` because it represents 3 elements with the last list element potentially being Nil.

Insertion sort using pattern matching

```
def isort(xs: List[Int]): List[Int] = xs match
    case List() => List()
    case y :: ys => insert(y,isort(ys)) //recursively sort tail of list

def insert(x: Int, xs: List[Int]): List[Int] = xs match
    case List() =>
    case y :: ys =>
        if x < y then x :: xs else y :: insert(x,ys) //if x < y then we make x the first ele
```

Enums

$4 + 4 = \sum_{n=i}$