Assignment 1 – Introduction

Advanced programming paradigms

In this serie, you will start by exploring the various tools at your disposal for developing functional programms. You will also be able to check that your tools have been installed correctly. In the second part of the serie, you will apply some of the knowledge you got during the first lesson in a practical exercise.

Question 1 – *Using ScalaIDE*

(a) Launch the ScalaIDE and create a new Scala project using *File → New → Scala Project*.

(b) Within ScalaIDE, you can develop "normal" Scala code with classes etc... but this will come later on.

At the moment, we will focus on the usage of *worksheets*. Worksheets are similar to the REPL, except that you have access to multiple commands at the same time. Evaluation takes place every time you save the file and the result of the evaluation is displayed on the right-hand side of the screen (as depicted below):

```
def foo = 5.0                          //> foo: => Double
def bar = 3                            //> bar: => Int
def sum(x: Double, y: Double) = x + y  //> sum: (x: Double, y: Double)Double
```

Note that the evaluation results are actually comments in Scala (the rule for comments are similar to C++ or Java). If required, you can force the evaluation of the worksheet by pressing the keys Ctrl + Shift + B together.

(c) Add a new worksheet (shortcut is Alt + Shift + N) called `FirstSteps`. Note that you can safely ignore the `object` declaration at the top of the screen now, it will be explained later.

(d) Run the examples from slide 71 in the REPL. Try to make yourself at ease by defining other things.

(e) In the worksheet, define a function that returns the square of a value. Check that your function works correctly by applying it to various values.

(f) Define another function that returns the 4th power of a value, using the `square` function you just defined.

(g) As you can see, the worksheet always returns the type that has been inferred for the expression you type or from the evaluation. What do you expect the worksheet to return for the following definition?

```
def bar(x: Int, y:Boolean) = "Hello"
```

....

....

....

⚠*Turn page →*

Question 2 – *Getting our hands dirty*

You are now asked to write a function to compute the square root of a number[1]. Its prototype should be

```scala
def sqrt(x: Double) : Double
```

1. The Newton's method

A typical numerical method to compute the zeroes (or roots) of a function is the Newton's method. Given a function f and its derivative f', we begin with a guess x_0 for the root. A better approximation x_1 of the root is then given by :

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{1}$$

The process is then repeated with the recursion equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2}$$

and stopped when the residual ϵ is small enough.

2. Application to the square root function

Let's say one wishes to compute[2] `sqrt(612)`. This is equivalent to $x^2 = 612$. The function to use in Newton's method is then $f(x) = x^2 - 612$. Its derivative is $f'(x) = 2x$. With an initial approximation of 10 (you can choose what you want here), the steps are then :

$$\begin{aligned}
x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} = 10 - \frac{10^2 - 612}{2 \cdot 10} = 35.6 \\
x_2 &= x_1 - \frac{f(x_1)}{f'(x_1)} = 35.6 - \frac{35.6^2 - 612}{2 \cdot 35.6} = 26.3955\ldots \\
&\vdots \\
x_5 &= 24.73863375\ldots
\end{aligned}$$

3. Implementation

As you can see, with only five steps the solution is already accurate to more than five decimal places (all the decimals written are correct). With the help of recursion, you now have to implement this method for computing square roots.

(a) Create a new worksheet in Eclipse to write your code for this assignment.

(b) Define a function `isGoodEnough` that determines if your solution is good enough. You solution can be considered good enough for example when $\epsilon < 0.0001$. For this part, you need to compute an absolute value function.

(c) Define another function, called `improve`, to compute the value of x_{n+1}, given the current approximated value and the value of x.

(d) Using the previously defined functions, define the `sqrt` method. Please note that you can add other functions if you need to!

(e) Test your method and check your results.

(f) *[Optional]* Implement the cubic root using the same approach and check your results.

[1]This exercise is originally from the SICP

[2]Example from http://http://en.wikipedia.org/wiki/Newton's_method

Assignment 2 – Higher-order functions

Advanced programming paradigms

In this assignment, you will work with tail recursion and higher-order functions.

Question 1 – *Tail recursion*

(a) Define a tail-recursive version of `fact`

(b) The Fibonacci function is defined as follows:

$$fib(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } x = 1 \\ fib(x-1) + fib(x-2), & \text{otherwise} \end{cases}$$

You are asked to write two implementation of this function.

1) The first one using the definition above.
2) A second solution using tail recursion. For this one, do not forget to check that your function is tail-recursive with the `@tailrec` annotation[1].

Question 2 – *Higher-order functions*

(a) The `sum` function we defined during the course and that computes

$$\sum_{i=a}^{b} f(x)$$

uses a linear recursion. Can you transform it to a tail recursion by filling the `???` hereafter?

```scala
def sum(f: Int => Int, a: Int, b: Int) = {
  def iter(a: Int, acc: Int): Int = {
    if (???) ???
    else iter(???, ???)
  }
  iter(???, ???)
}
```

Question 3 – *Currying*

(a) Using the `sum` function as a source of inspiration, write a function `product` that computes the product of the values of a function for the integers in a given interval, i.e.

$$\prod_{i=a}^{b} f(i)$$

Make sure that this function is in its curried form.

(b) Write `factorial` in terms of the function `product` that you defined in part *a*.

(c) Write a more general function that generalizes both `sum` and `product`. This done, provide a new implementation of `sum`, resp. `product`, using that new function.

[1]For which you have to import `scala.annotation.tailrec`

Assignment 3 – Functional data structures

Advanced programming paradigms

In this assignment, you will add several methods to the `IntSet` that we discussed in class. You will be able to practice object-oriented programming mixed with functional-programming, a first example of multi-paradigm programming. Let us first consider the following code, which corresponds to the implementation of the `IntSet` that was given during the lecture:

```scala
abstract class IntSet() {
  def add(x: Int): IntSet
  def contains(x: Int): Boolean
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def add(x: Int): IntSet = {
    if (x < elem) new NonEmpty(elem, left add x, right)
    else if (x > elem) new NonEmpty(elem, left, right add x)
    else this
  }

  def contains(x: Int): Boolean = {
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  }
}

object Empty extends IntSet() {
  def contains(x: Int): Boolean = false
  def add(x: Int): IntSet = new NonEmpty(x, Empty, Empty)
}
```

Question 1 – *Displaying the content of the set*

To help visualizing and debugging of your code, we will start by devising a way to display a set.

(a) Begin by overriding the `toString` method for the two sub-classes so that the following code prints as follows:

```scala
println(Empty)                  // prints -
println(Empty.add(3))           // prints (-|3|-)
println(Empty.add(3).add(2))    // prints ((-|2|-)|3|-)
```

(b) In the second part of this exercise, we will not make a method only for printing a set but we will instead develop a more general method that we will then specialize later for displaying the content of a set.

The general method we will develop is called `foreach`. This is a standard method for collections in functional programming and its purpose is to apply a function to every element of the collection. The prototype of the method is as follows:

```scala
def foreach(f: Int => Unit): Unit
```

Please note that the `Unit` type declared here is a bit special. This type has a single value, called unit as well – denoted `()` –, which is used when the return value of a function is of no interest. Contrary to the Java `void` type, it has a value and **can** therefore be used in an expression. Test that your method is working as expected on a set `s` by calling

```
s.foreach(println)
```

(c) Apply the `foreach` method to display every element of the collection incremented by one. Each element should be separated by a comma (ignore the fact that the last element shows a comma as well). For instance:

```
(Empty.add(3).add(2).add(6).add(1)) foreach (...)
```

should print `"4, 3, 2, 7,"`. ⚠ You should be able to write this code as an one-liner!

(d) In the last question, why is the result `"4, 3, 2, 7, "` and not `"4, 3, 7, 2, "`, which corresponds to the insert order?

. . . .

. . . .

. . . .

. . . .

Question 2 – *Union and intersection of sets*

Very common operations in set theory consist in computing the union and intersection of two sets. You will now implement those methods on `IntSet`.

(a) Add a new method called `union` for forming the union of two sets. Modify the abstract class accordingly. The prototype of the method is as follows:

```
def union(other: IntSet): IntSet
```

(b) Add a new method to compute the intersection of two integer sets. The intersection of two sets should comprise all the elements of the first set which also belong to the second set.

Question 3 – *Adding new operators*

We will now experiment with the implementation of new operators on the set we have defined.

(a) Start by adding to `IntSet` the following method:

```
def excl(x: Int): IntSet
```

which should return the given set without the element `x`. It might be useful (but it is not required) to implement the following helper method as well:

```
def isEmpty(): Boolean
```

(b) Make the necessary changes to your code so that it becomes possible to write the following expression (express you solution in terms of the existing methods):

```
val o1 = Empty + 3 + 4 + 12 + 5
val o2 = (o1 - 3 - 4)
```

The `toString` method of the resulting set should yield:

`(-|5|(-|12|-))`

ASSIGNMENT 4 – PATTERN MATCHING & LISTS

Advanced programming paradigms

In this assignment, you will work with pattern matching in the context of expressions evaluations and lists. In the last two exercises, you will implement several handy functions to work with lists. Even though these functions are already present in the Scala `List` library, implementing them is a good practice to understand how they work.

Question 1 – *Working with expressions*

Here is the code of the expression interpreter shown in class:

```scala
sealed abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Product(e1: Expr, e2: Expr) extends Expr

def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
  case Product(e1, e2) => eval(e1) * eval(e2)
}
```

(a) Add to this expression interpreter the product `x*y` operation. *Hint:* pattern matching with constructor pattern also works recursively.

(b) In addition, change your show function so that it also deals with products.

⚠ Pay attention you get operator precedence right and that you use as few parentheses as possible when showing your expression.

(c) When you are done, check your code with the following snippet:

```scala
val expr0 = Sum(Product(Number(2), Number(3)), Number(4))
println("Expr0: " + show(expr0))
assert(eval(expr0) == 10)

val expr1 = Product(Number(4), Number(12))
println("Expr1: " + show(expr1))
assert(eval(expr1) == 48)

val expr2 = Product(Sum(Number(2), Number(3)), Number(4))
println("Expr2: " + show(expr2))
assert(eval(expr2) == 20)

val expr3 = Product(Number(2), Sum(Number(3), Number(4)))
println("Expr3: " + show(expr3))
assert(eval(expr3) == 14)
```

It should produce no assertion error and display the following result (pay attention to the parentheses for the product in `Expr3` and `Expr4`):

```
Expr0: 2*3+4
Expr1: 4*12
Expr2: (2+3)*4
Expr3: 2*(3+4)
```

Question 2 – *Defining trees*

We now propose a similar approach for modelling trees, as follows:

```scala
sealed abstract class BinaryTree
case class Leaf(value: Int) extends BinaryTree
case class Node(left: BinaryTree, right: BinaryTree) extends BinaryTree
```

Starting with this code,

(a) write a function to compute the sum of the **leaves** of the tree.

(b) write a function to find the smallest element of the tree. Please not that we are not using a *binary-sorted tree* here, which means you have to check every node of the tree.

Question 3 – *Lists functions*

(a) In order to apply pattern matching and exercise the way of thinking with lists, you have to implement several functions on lists. Please note that your functions should work on lists of arbitrary types.

- `last`, which returns the last element of a list;
- `init`, which returns a list of every element but the last;
- `reverse`, which returns the list with its elements in the reversed order;
- `concat`, which concatenates two lists together;
- `take(n)`, which returns the first *n* elements of the list;
- `drop(n)`, which returns the list without its first *n* elements. If $n > length(list)$, then `Nil` should be returned;
- `apply(n)`, a functions that returns the *nth* element of a list. Note that the first element of a list is a position 0.

(b) What is the complexity of

1) `last`

1) ______________________

2) `concat`

2) ______________________

3) `reverse`

3) ______________________

Question 4 – *Predicates on lists*

(a) Define the function `any` which should have the following prototype[1]:

```scala
def any[T](p: T => Boolean)(l: List[T]): Boolean
```

This function should return `true` if any element of the list satisfies the predicate.

(b) Define then the function `every` which controls if *every* element of a list satisfy this predictate. Use pattern matching.

[1]This exercise is taken from the *Programmation 4* course given by M. Odersky at EPFL

ASSIGNMENT 5A – HOF ON LISTS

Advanced programming paradigms

In this assignment, you will apply different higher-order functions on lists to gain experience in those typical functional-oriented operations.

Question 1 – *Higher-order functions on lists*

Using only the `zip` and `map` higher-order functions, define the following functions that:

(a) Returns a list of the length of each string of a list
Example:
`lengthStrings(List("How","long","are","we?"))` returns `List(3,4,3,3)`

(b) Produces a list with n identical elements of arbitrary type (don't use the `fill` method!)
Examples:
`dup("foo", 5)` returns `List("foo", "foo", "foo", "foo", "foo")`
`dup(List(1,2,3), 2)` returns `List(List(1,2,3), List(1,2,3))`

(c) Multiplies element-wise two lists of values and create a new list
Example:
`dot(List(1,2,3), List(2,4,3))` returns `List(2,8,9)`

Question 2 – *Folding functions on lists*

Now using folding (right or left) define a function that:

(a) Determines if all logical values in a non-empty list are `true`.
Examples:
`areTrue(List(true, true, false))` returns `false`
`areTrue(List(true, true, true))` returns `true`

(b) Determine the total length of the strings in a list
Example:
`lString(List("Folding", "is", "fun"))` returns 12

(c) Returns the longest string of a list as well as its size
Example:
`longest(List("What", "is", "the", "longest?"))` returns `(8, longest?)`

(d) Decide if a value is an element of a list of an arbitrary type
Examples:
`isPresent(List(1,2,3,4), 5)` returns `false`
`isPresent(List(1,2,3,4), 3)` returns `true`

(e) Flatten a nested list structure of any type.
Example:
`flattenList(List(List(1,1), 2, List(3, List(5, 8))))` returns `List(1,1,2,3,5,8)`

Assignment 5b – Sequence comprehensions

Advanced programming paradigms

In this assignment, you will apply your new knowledge on tuples and use sequence comprehensions (*for*) to produce an exhaustive search for postcards and to work with the n queens problem.

Question 1 – *Tuples*

This is the code given in class for returning the sums of the couple of integers under a certain value that are prime:

```
def primeSum(max: Int): List[(Int, Int)] =
  for {
    i <- (1 to max).toList
    j <- (1 to max).toList
    if (isPrime(i + j))
  } yield (i, j)

def isPrime(i: Int): Boolean =
  i match {
    case i if i <= 1 => false
    case 2 => false
    case _ => !(2 to (i - 1)).exists(x => i % x == 0)
  }
```

The problem of this code is that the `primeSum` function returns values twice. For instance, it returns `(1,2)` but also `(2,1)`. Because those values are only permuted, we would like to remove redundant pairs. Without changing the existing code, devise a function that removes the existing permutations from a similar list.

(a) Using pattern matching with tuples and recursion

(b) Using folding with tuples

Question 2 – *Sending postcards*

In this exercise, you will use for comprehensions for generating postcards when you are travelling. You are given the following code that codes defines the city you visit, the person you want to send the cards to and the travellers that are sending postcards.

```
val cities = List("Paris", "London", "Berlin", "Lausanne")
val relatives = List("Grandma", "Grandpa", "Aunt Lottie", "Dad")
val travellers = List("Pierre-Andre", "Rachel")
```

(a) In each city you visit, each travellers send a postcard to all their relatives. Because the task is tedious, you are asked to write a program that generates the text for the postcards as follows:

```
    Dear Grandma, Wish you were here in Paris! Love, Pierre-Andre
    Dear Grandma, Wish you were here in London! Love, Pierre-Andre
    ...
    Dear Grandpa, Wish you were here in Paris! Love, Pierre-Andre
    Dear Grandpa, Wish you were here in London! Love, Pierre-Andre
    ...
    Dear Grandma, Wish you were here in Paris! Love, Rachel
    Dear Grandma, Wish you were here in London! Love, Rachel
    ...
    Dear Dad, Wish you were here in Lausanne! Love, Rachel
```

(b) How do you modify your code in order to send postcards only to the family members whose name start with the letter `G`? You answer can (and should) be written in one line.

Question 3 – *Printing the n queens solutions*

A possible solution to the n-queen problem that your had to analyze is as follows:

```scala
def queens(n: Int): List[List[(Int, Int)]] = {
  def placeQueens(k: Int): List[List[(Int, Int)]] =
    if (k == 0)
      List(List())
    else
      for {
        queens <- placeQueens(k - 1)
        column <- 1 to n
        queen = (k, column)
        if isSafe(queen, queens)
      } yield queen :: queens

  placeQueens(n)
}

def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
  queens forall (q => !inCheck(queen, q))

def inCheck(q1: (Int, Int), q2: (Int, Int)) =
    q1._1 == q2._1 || // same row
    q1._2 == q2._2 || // same column
    (q1._1 - q2._1).abs == (q1._2 - q2._2).abs // on diagonal
```

Using for comprehensions, write a function `def printChessBoard(...)` that returns a string containing all the solutions for a given n. For instance,

```scala
println(printChessBoard(queens(4)))
```

should display the following on the console:

```
Solution 1:
|_|♛|_|_|
|_|_|_|♛|
|♛|_|_|_|
|_|_|♛|_|

Solution 2:
|_|_|♛|_|
|♛|_|_|_|
|_|_|_|♛|
|_|♛|_|_|
```

Remark: Note that the Unicode character for the black queen used in the above example is `\u265b`.

Assignment 7 – Streams, types and lazyness

Advanced programming paradigms

Question 1 – *A covariant stack*

In this first question, you are given the following trait:

```scala
trait Stack[A] {
  def push(elem: A): ...
  def top: A
  def pop: Stack[A]
}
```

Starting with this code, you have to define a covariant definition of this stack with its implementation. For this, you will follow the idea we used in assignment 2 for the integer set.

(a) Begin by declaring two case classes, `EmptyStack` and `ElemStack`. The former is used to model an empty stack and the later is used to store elements. Do not forget we are only working with immutable data structures!
Hint: the constructor of the `ElemStack` requires to take an element as well as another `Stack`.

(b) Implement the methods in those classes.

(c) Considering that the same person implements everything, where is the best location to implement the `push` method? Why?

. . . .

. . . .

. . . .

. . . .

(d) When you are done, the following code should execute without failures

```scala
// Construction, pop and toString
val a = EmptyStack().push("hello").push("world").push("it's fun").pop
assert(a.toString() == "world,hello,EmptyStack()")

// Getting top
val b = EmptyStack().push(1).push(3)
assert(b.top == 3)

// Variance checks
class Foo
class Bar extends Foo
val c: Stack[Bar] = EmptyStack().push(new Bar()).push(new Bar())
assert(c.top.isInstanceOf[Bar] == true)
assert(c.top.isInstanceOf[Foo] == true)

// Variance check 2
val d: Stack[Foo] = EmptyStack().push(new Bar()).push(new Bar())
assert(d.top.isInstanceOf[Foo])
```

Question 2 – *The Fibonacci sequence using infinite streams*[1]

To get used to infinite sequences, you will define the infinite stream of the Fibonacci sequence. In this sequence, the first two elements are 0 and 1. Each subsequent element is obtained by summing the two preceding elements.

To define this stream, start by writing the function `addStream`, which takes two integer streams in argument and returns a new stream whose elements are the sum of the elements of the two input streams. Its prototype is as follows:

```
def addStream(s1: Stream[int], s2: Stream[int]): Stream[int]
```

Using this function, the definition of the Fibonacci sequence takes a single line.

Question 3 – *Streams of prime numbers*

(a) Define an stream for integers which start at a given value and that continues up to infinity.

(b) Use this `Stream` to define the sequence of the prime numbers, using the techniques known as the *Sieves of Eratosthenes*. This very ancient technique works as follows:

Start with an infinite sequence of integers, starting with 2:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15...
```

We then take the head of the list (which is **always** a prime number) and we eliminate all the multiples of this number. We obtain a new infinite sequence:

```
2 3 5 7 9 11 13 15 17...
```

We recursively apply this function to the rest of the list (here 3 is the new head), obtaining:

```
3 5 7 11 13 17...
```

ASSIGNMENT 8 – PARALLELISM

Advanced programming paradigms

Question 1 – *Parallel collections*

In this first exercise, you will experiment on how parallel collections can be easily used to improve the performance on standard operations in FP[1].

Sometimes, a program needs to integrate a function (i.e., calculate the area under a curve). It might be able to use a formula for the integral, but doing so is not always convenient, or even possible. An easy alternative approach is to approximate the curve with straight line segments and calculate an estimate of the area from them. This is called the *trapezoidal rule*, which is a technique used to approximate the value of a definite integral $\int_a^b f(x)dx$.

To compute the value of the integral numerically, we can apply the following algorithm (by supposing $f(x)$ is continuous on $[a, b]$).

1. Start by dividing $[a, b]$ into n sub intervals of equal length $\Delta = \frac{b-a}{n}$.
2. Compute $f(x)$ at each point to obtain $y_0 = f(x_0), y_1 = f(x_1), \ldots, y_n = f(x_n)$.
3. As straight lines are formed between the points (x_{i-1}, y_{i-1}) and (x_i, y_i) for $1 \leq i \leq n$, we can approximate the integral using n trapezoids

The following figure illustrates how the trapezoidal method can be applied to a general function:

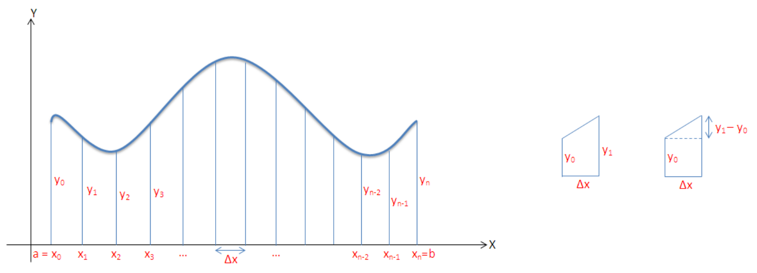

Figure 1 – The trapezoidal method in general and trapezoid area computation

To compute one of the trapezoids area, we can use the following formula:

$$A = y_0\Delta x + \frac{1}{2}(y_1 - y_0)\Delta x$$
$$A = \frac{(y_0 + y_1)\Delta x}{2}$$

In the end, adding the area of n trapezoids is an approximation of the integral

$$\frac{(y_0 + y_1)\Delta x}{2} + \frac{(y_1 + y_2)\Delta x}{2} + \ldots + \frac{(y_{n-1} + y_n)\Delta x}{2} \approx \int_a^b f(x)dx \tag{1}$$

If we have very many divisions to compute the integral (to have a precise values for instance), it might be interesting to compute the y_i values in parallel. However, even though this equation enables parallel computation, it requires to compute most values twice (y_1 for instance). As it would be a bit silly to try to accelerate an inefficient computation, we can rewrite (1) as follows:

$$\approx \frac{\Delta}{2}(y_0 + 2y_1 + \ldots + 2y_{n-1} + y_n) \tag{2}$$
$$\approx \frac{\Delta}{2}(y_0 + y_n + 2\sum_{i=1}^{n-1} y_i) \tag{3}$$

[1]This exercise is adapted from the *SIGCSE 2013 Workshop on Scala*.

This latest solution requires now to compute each y_i once. As a result, the sum can be easily executed in parallel, a feature that we will leverage in the exercise.

(a) Define a method called `integrate` to compute the integral of a general function (which can be defined as a parameter). This function should use the trapezoidal integration technique described earlier. The complete prototype of the function should be as follows:

```
def integrate(a: Double, b: Double, nIntervals: Int, f: (Double => Double))
```

Check that your function returns the proper result. For instance, $\int_1^2 sin(x)\ dx \approx 0.956449$

(b) The function you integrated above was composed of a single mathematical function $sin(x)$. Scala offers a way to compose (in a mathematical sense) functions. Using the `compose` operator, compute $\int_0^1 sin(cos(x))\ dx$. The result should be ≈ 0.738643.

(c) Measure the time[2] required for computing a result using 20^6 intervals.

(d) The performance of the method you have developed so far is not optimal. In fact, applying various operations in sequence to a list is not really efficient because all the intermediary result have to be computed and stored. Take for instance the following case:

```
(1 to 10e6.toInt).
  map(Math.sin(_)).
  map(_ * 12).
  map(Math.pow(_, 4))
```

This requires to create 3 different collections and apply the operation on each intermediary result. As we did with the `Stream` class, it is possible to create *lazy collections* with the `view` method. Thus, writing:

```
(1 to 10e6.toInt).
  view.
  map(Math.sin(_)).
  map(_ * 12).
  map(Math.pow(_, 4))
```

consumes far less memory and goes almost two times faster. In addition, if all the results are not required, the results are not computed. Thus, in the second case, if we only require the first element of the result, only a single value will be computed!

Apply this technique to the integration method and measure the time required to compute a result. Why is it faster?

....

....

....

(e) Implement now a new version of the integration using *lazy parallel collections*. Please note that the change on the collections themselves should be a matter of adding a `.par` somewhere. Measure the time taken to compute the numerical integral of the function using both the parallel and sequential collections. What speed-up do you get on your machine in the parallel version compared to the view version? Does it correspond to the speed-up you were expecting considering the number of logical processors your machine possesses?

[2] To measure time you can download the `utils` package on the Moodle website

. . . .

. . . .

. . . .

. . . .

Question 2 – *Using futures in a real-world example*

A typical use case for futures is to asynchronously read results from the Web. In this exercise, you will use futures to read values from two public REST APIs to know the exchange rate for Bitcoins in Euros.

(a) Write a method returning a `Future[String]` to read the content of a web page. Note that you can read data from a URL simply by using the `Source.fromURL(...).mkString` method. To make it work, import `scala.io.Source`.

(b) Access the data from the following URL `https://api.bitcoinaverage.com/ticker/USD/` which provides the exchange rate from Bitcoins to USD. The returned `String` is formatted in JSON.

(c) Access the data from the URL `https://api.bitcoinaverage.com/ticker/EUR/` which gives the exchange rate from Bitcoins to €. This is also formatted as JSON.

(d) To parse JSON in Scala[3], download the two JAR files from the Moodle website (`lift-json_XXX.jar` and `paranamer-2.1.jar`). You need then to include those JAR into your Eclipse workspace, *Right-click on your project → Build Path → Configure build path → Add JAR . . .*). When you are done, the following program should execute without problem:

```scala
import net.liftweb.json.DefaultFormats
import net.liftweb.json.parse

object JSONSimple extends App {
  implicit val formats = DefaultFormats

  // Sample JSON data
  def bitcoin_json() = {
    """{
      "ask": 458.05,
      "bid": 456.59,
      "last": 456.41,
      "timestamp": "Mon, 14 Apr 2014 12:09:35 -0000",
      "total_vol": 93280.72
      }"""
  }
  val jVal = parse(bitcoin_json) // JSON to val

  // Primitive extraction a la XPATH
  val a = (jVal \ "bid").extract[Double]
  println("Extraction of 'bid': " + a)
}
```

(e) Using futures composition with comprehensions, parse the results from the two URLs and display the USD vs EUR rate.

[3] There many libraries to parse JSON in Scala. The one you will be using here comes from the *Lift!* framework which is typically used for creating web services. In this exercise, you will be using only the JSON library part of that framework. The library allows for several types of JSON parsing, using extraction for case classes, XPATH style extraction . . . . It also enables the creation of JSON data from Scala classes. The example above is very simplistic.

ASSIGNMENT 9 – DSLs

Advanced programming paradigms

Question 1 – *Implicit conversions*

You will develop a class `TestConv` that enables easy temperature conversions. It uses a a common type for every temperature – `Temperature` – which should be declared `abstract` and `sealed` (all the sub-classes declared in this source file are the only subclasses allowed). Two sub-types of this class are `Celsius` and `Kelvin`.

(a) Write the required *implicit conversions* and the required code to allow the following code to run:

```
val a: Celsius = 30
val b: Kelvin = 30
val c: Kelvin = Celsius(10)
val d: Celsius = c
val e: Temperature = d

println(a) // Should print "30° C"
println(b) // Should print "30° K"
```

(b) What will you get on the console if you try to print `e`?

. . . .

(c) Why is it interesting to have the `Temperature` class? Explain!

. . . .

. . . .

. . . .

. . . .

Question 2 – *Interactive lab*

In this interactive lab, we will discover together how a classical example of interactive drawing program can be adapted to express more simply complex ideas thanks to a an internal DSL[1].

(a) In this first part, we will discover the example code and see how to make it work

1. In the Scala IDE, make a new Scala project lab1 with a Scala application `Lab1` and paste in all this code. Run the program. What happens? Why? (Look at code in the `App` object.)
2. Have a look at the code. You'll see a bunch of geometric shapes: `Circle`, `Rectangle`, `Arrow`, all of which extend `Drawable`. There is also a class `Point` for specifying points with (x, y) values.
3. Next, there is a `Drawing` – a Swing component to which you add `Drawable` objects.
4. Then there are a bunch of effects. An effect is something that happens as time elapses—moving a shape, making it visible or invisible, etc. The effects are pretty simple—their act method is called many times, and it does something, such as moving the center (`MoveEffect`) or changing the transparency (`HideEffect`).
5. More interesting are the effect combinators. One wants to say *"Do these two effects together, and then do the other effect."* When `e1` and `e2` are effects, then you can make a `TogetherEffect(e1, e2)` that runs them in parallel, and an `InOrderEffect(e1, e2)` that runs `e1` and then `e2`, and also a `BackwardsEffect(e1)` where time runs backwards.
6. So, now put these to use. Make it so that `c1` moves as before, and when it is done, then `c2` moves to `new Point(200, 200)` in 6000 milliseconds.
7. Make it so that, as `c2` moves, it also hides in 3000 milliseconds.

[1]This interactive lab was given by C. Horstmann during his invited lecture in 2015

(b) In this second part, we will start making our *Effect* DSL.

1. That API is a mess. We really want to say

```
e1 followedBy (e2 and e3)
```

2. Make it so. Simply define methods `followedBy` and `and` in the `Effect` class that return an `InOrderEffect` or a `TogetherEffect`.
3. Also define a method `reversed` that makes a `ReverseEffect` and try out

```
e1 followedBy (e2 and (e3 followedBy e3 reversed))
```

4. What does it do?
5. Maybe that's nicer with operators? Make it so one can write

```
e1 ==> (e2 || (e3 ==> -e3))
```

6. How did you do that?

(c) Code blocks

1. What if we want to make another change to those shapes? Maybe we want a circle to grow. Sure, one could write a `GrowEffect`. But wouldn't it be nicer if we could just specify the grow behavior in a code block? Like

```
val e = update(2000) { t => c2.radius = 30 + 20 * t }
```

to indicate that `c2` should have its radius changed from 30 to 50 in 2000 milliseconds.

2. So, we need an `UpdateEffect`. It should take
   - A duration
   - A block of code for updating, with a `Double` parameter and `Unit` result

   Its `act` method simply calls the code block with `completion(t)`, which ranges from 0 to 1 as the timer tick ranges from 0 to the duration of the effect. Implement the class. Then implement a curried `update` method (for simplicity, in the `Lab1` object) that makes an instance, given an `Int` and a code block. Then start the effect `e` above.
3. Explain what happens if you try

```
val e = update(2000) { c2.radius = 30 + 20 * _ }
```

(d) Implicit conversions

1. We want to repeat an effect `n` times. That's easy, thanks to the miracle of recursion:

```
abstract class Effect ... {
  ...
  def times(n: Int): Effect = {
    if (n == 1) this else new InOrderEffect(this, times(n - 1))
  }
}
```

   Add that method, and then change the `Lab1` object to call `d.start(e1 times 3)`
   What happens?
2. What happens when you try

```
d.start(3 times e1)
```

Why?

3. Ok, that can't work — `times` isn't a method of `Int`. So that's where implicit conversions come in. We need to convert `Int` to some object, say `EffectInt`, with a `times` method. Make such a class and method, and then try out

```
d.start(new EffectInt(3) times e1)
```

4. Sure, that works, but it's ugly from the point of view of a DSL. Make an implicit conversion from `Int` to `EffectInt`. Just place it inside the `Lab1` object. Then try

```
d.start(3 times e1)
```

and rejoice.

(e) Implicit parameters

1. Right now, the code for making an arrow between two `Drawable` is

```
d += new Arrow(c1, c2)
```

2. Really, in a DSL, we'd like to say

```
c1 --> c2
```

3. What about the `d+=` ?
4. That's boring — of course we need to add the arrow to the component so that it gets painted.
5. Relieving boredom is what implicit parameters are for. Define a `->` method on the `Drawable` class with a regular parameter `to: Drawable` and an implicit parameter `d` of type `Drawing`. Make it call `d += new Arrow(this, to)`.
6. Now replace `d += new Arrow(c1, c2)` with `c1 -> c2` in `Lab1`. What happens?
7. That couldn't have worked. There is no implicit `Drawing` anywhere. Add `implicit` before the declaration of the `Drawing` instance `d` in `Lab1`. Now what happens?