Programming with Scala: Language Exploration

Bhim Upadhyaya ©2017

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Computing

The Oxford English Dictionary (OED) defines computing as "the use of operation of computers"; similarly, computation is defined as "the action of mathematical calculation." In daily life, we often find these words being used interchangeably even though scientific community makes distinction. Let's first analyze computation as it came first in the human civilization, formally with the invention of numbers. But it is quite self-evident that humans performed computation before inventing numbers as there should be a though process before finding suitable symbols for that thought process. This kind of thought process is likely to be available in other *Mammalias* as well as in some other *Classes*, categorized using traditional biological taxonomy.

Let's take two examples to illustrate computation: $1+1=2$ and $13+29=42$. Now, let's ask ourselves these questions: *What percentage of world population can perform first addition? What percentage of the world population can perform second addition without using a calculating machine What percentage of world population can perform second addition using a calculating machine?* We should not be surprised if the answer to our first question is not 100%. The United Nations' data show that answers to our second and third questions are not 100% [UNL13].

Analyzing further in the same direction, there are many more questions to be asked including: *How long it took us to recognize real world objects? How long it took us to take instructions (both in the form of signs and spoken language) from elders and perform the addition task for the first time in our lives? How long it took us to recognize written alphabets and numerals? How long it took us to perform written addition? How long it took human kind to be in this state of mind, which allows one to instruct and another to follow instructions and perform actions?* These questions might look a bit overwhelming and unnecessary at first, but these and many other similar questions govern our learning life cycles.

Now, let's take slightly different example to set stage for our Scala lessons.

SN	**Hiearchy**	**Human**	**Dog**	**Domestic Pigeon**	**Cat**
1	*Kingdom*	Animalia	Animalia	Animalia	Animalia
2	*Phylum*	Chordata	Chordata	Chordata	Chordata
3	*Class*	Mammalia	Mammalia	Aves	Mammalia
4	*Order*	Primates	Carnivora	Columbiformes	Carnivora
5	*Family*	Hominidae	Canidae	Columbidae	Felidae
6	*Genus*	Homo	Canis	Columba	Felis
7	*Species*	H. Sapiens	C. lupus	C. livia	F. catus

Table 1.1: Sample Biological Taxonomy Data

This too might look counter intuitive initially but we will write a Scala program for this later in this chapter. Table 1 shows sample biological categorization of human, dog, domestic pigeon, and cat. Here are some of the questions. *Is it a computational problem? Do we have sufficient information to decide whether it is a computational problem?*

Let's say, we are asked to build a dictionary or an information base that can be referred to get information. Now, it is fairly convenient to decide whether it is a computational problem if we have computing background. But this might be still confusing if we do not have any idea about computing, because computation cannot be seen on the surface. Even Google search may not look like a computational problem on the surface as we can't see regular calculations. In fact, Google search is a complex computation.

Assuming we don't have any knowledge of computing as defined by OED. Probably it is fair to say that all the human beings search at least one item in their life. When we are searching something, our mind performs computation. We might need to locate, count, or categorize items. Locating something might involve counting. For example, if we have to locate a book in another room, then we have to cross at least one door, assuming these are regular rooms in a regular house. Since we have enormous practice going from one room to another room in our lives, we might be performing the computation even without realizing it. Now, let's think about what it takes to train an infant as he/she grows to perform the same task. Does the infant need to learn how to count in order to perform this task? Probably the answer is yes. And it might take years to train the infant. Learning computing is not much different from this infant's training. The major difference is age. And of course, infants too can start learning computing these days.

We know how hard it is to live our lives without using any tool. Even in stone age, our ancestors used some sort of tools: a stone, a stick, or a little more sophisticate tool. Now, all of us, we know why we need tools. Also we know that it is not the same tool that can be utilized to solve every problem in our lives. This is true in the case of computational tools as well. Since this book deals with a particular programming language, Scala, in details, let's be concrete and say that this is true for programming languages as well. Programming languages are parts of computational tools.

Now we have some ideas about computation. Let's ask another question: *can every computational problem be computed?*. Well, there are several university level courses dedicated to answer this question. For now, we focus on our two problems—addition and tiny information base for biological taxonomy. The first one can certainly be computed. We limited the scope of second and made it viable for computing. Please note that we did not go for genomics, which requires enormous computing power.

In the following section, we discuss the basics of computing tools, called computers.

1.1 Introduction to Computers

Computers are the tools that we can use to perform some computations and come in various shape and size. There are hundreds of companies around the world that manufacture varieties of computers and computer parts. Generally a computationally useful computer has two categories of components—hardware and software. Hardware consumes energy and performs computations whereas software contains the logic for operations. It is the software that instructs the hardware in order to achieve a computational goal. A computational goal can can be as primitive as inverting a digit, converting 0 to 1 and vice-versa. In this book, we will learn a programming language that helps us to instruct computers in order to achieve one or more computational goals. Clearly it is a software component that helps us to create other software components.

1.1.1 Basic Components

Digital computers have the following basic components:

- Memory Unit
- Processing Unit
- Storage Unit

- Input Device
- Output Device

Almost every digital computing machine has some sort of memory. For example, if we are performing addition mentioned earlier, $1 + 1 = 2$, using a digital calculator, it remembers at least three different items: digit 1 (first operand), operation + (operator), and digit 1 (second operand). A typical laptop, say a laptop from *One Laptop per Child*, has much more memory than a typical calculator [OLC17]. The reason is that a laptop has to hold much more information and has to perform much more sophisticated operations than a typical calculator. And things might be different if we are referring to a scientific calculator. For now, we stick with a simple calculator that performs addition, subtraction, multiplication, and division.

In case of our calculator, we need a unit that performs the addition operation. The unit that performs this kind of operations is called *Processing Unit*. In case of digital computers, every high level operation or computational goal like opening a file or googling a word is eventually represented with 1s and 0s, these are the only two signals a digital computer understands. Interestingly, this transformation is a complex process and is studied as a computer engineering degree in traditional universities. From this course's perspective, let's remember the fact that every program we write will be eventually processed by a processing unit. A common terminology used for such a processing unit is *Central Processing Unit* as there are other processing units in a typical computer. For example, a keyboard has a processor to process keyboard inputs.

Generally storage unit refers to permanent storage devices; these devices can retain data even after power is switched off. Large files or data go to storage devices and fetched to memory as per computational needs. Storage devices can be a small stick like a flash drive, or hard drive like in a laptop or a desktop, or array of disks. In case of cloud computing storage can be as big as hundreds of data centers around the world.

There are different ways to provide data to a computational device. Typing, speaking, and touching are common methods to provide input. Also different types of sensors can be utilized to provide input data. For example, wind direction sensor can generate and supply data to a computational device. If we we are typing, keyboard is a common way to provide input. Keys are continuously scanned for possible input. Microphones are common devices to capture voice signals. These analog signals are then converted to digital equivalent and fed to a computer. Please note that analog to digital converter itself does some sort of computation, in addition to signal conversion. So in a typical computer, there are numerous processing units, commonly called processors. For touch related input, a digital keyboard can be available; in some cases, it could just be a swipe.

Output devices present results of computations. Most often these results are visual and hence display devices are common output devices. Like computers, display devices can come in various shape and size. Some displays are boxed together with central processing units while others are packaged separately. Increasingly, displays devices have been using liquid crystal display (LCD) technology. LCD makes displays much lighter compared to older popular technology like cathode ray tube (CRT).

1.1.2 Operation

Computers operate in different modes as far as program execution is concerned. A program is a set of instructions, written in computer understandable form and mapped to one or more computation goals. If we are using portable computers, like laptop, most often the computation will be real time unless we instruct the machine to perform differently. But in a large farm of computers and processes mode of operation can be in a batch mode.

A batch mode is an operation mode which takes set of operations and executes in a non real time fashion. Also the results are often returned to the owner in a batch mode. This kind of operation is very common where there is massive data and real time processing is not feasible. For example, there is petabytes of data generated from Facebook and if we need to infer based on this large volume of data, real time processing based on state of the art computing technology is not practical. So there is a clear need for different mode of operation. Similarly, if certain high speed computing resources are in demand, we might need to provide time slots to each requester. This mode of operation is called time-sharing. In fact, batch mode of operation might utilize time sharing.

1.2 Operating Systems

An operating system (OS), as name suggests, is the software that operates hardware components. In order for us to execute our programs, we need an operating system. Our programs will be some of the software pieces that an operating system runs. Also if we need to access a service offered by an IO device then we have to request operating system for that service. One can think operating system as an administrator of the hardware resources. Operating system talks to various hardware components through drivers. Drivers are software programs that can process signals to and from hardware components.

A computer that is generally used by one person is called a personal computer (PC). PCs have the following popular operating systems:

- Linux family
- Mac OS family
- Windows family

Linux is the only open source operating system among the above three. Also it is the most popular. There are many dialects of Linux including Ubuntu, Suse, and Fedora core. Early days of Linux were marked by difficulty to install and manage. But now it is very different and Linux is as convenient as Windows operating system. Most of websites are hosted in a server that runs Linux operating system. Linux was inspired by Unix operating system, a popular enterprise operating system before Linux.

Mac OS is the operating system for Macintosh computers by Apple. Apple personal computers are known for their reliability and for their aesthetic value. Steve Jobs, one of the founders of Apple, talks about taste for computing and claims that Apple product provide good taste of computing. Remember we discussed about computing and computation as defined in OED, and now we are discussing about taste of computing. It should be noted that programming languages are developed in that fashion too. Like Linux, Mac OS has it roots to Unix and if we know of these three, working with other two is fairly convenient.

Generally Windows operating systems is regarded as easy operating system. It is a common operating system for personal computing devices. Some of the hardware manufacturers have contract with Microsoft, the company behind Windows, that allows them to sale Windows along with the new hardware. For example, if we buy IBM ThinkPad, now Lenovo ThinkPad, we are most likely to get Windows OS.

1.3 Programming Languages

Now we are getting closer to what we will be learning in this book. Programming languages can be categorized into two main categories:

- Low Level Programming Languages: Machine, Assembly
- High Level Programming Languages: Pascal, C, Java, Scala

```
...
SUB AX, AX
MOV ES, AX
SUB BH, BH
SHL BX,1
...
```

Figure 1.1: Sample Assembly Language Code Snippet

Early digital computers involved relay based switching in order to carry out a typical computation. This means programming in 1s and 0s. The next level of programming language was assembly language. Assembly language uses mnemonics; programs become longer and harder to comprehend.

The code listing in Figure 1.1 has register level operations. A register is one of the smallest data holding places in a computer. The width of registers partially determines computational capacity of a computer. A computational capacity is often related to the speed of a computer. So assembly language programming means register level programming, which some programmers find interesting. It gives more power as a programmer can directly access the content of a register. With this power comes a lot of responsibility on programmer's side.

A programmer has to take care of fine grained computational details. For example, if a programmer is writing a program to subtract two numbers and if these numbers do not fit to the width of one register then it is pro-grammar's responsibility to store and manage each digit of that integer. While performing subtraction, programmer is required to take care of carry overs as well. This was the life of a programmer before high level programming languages were created. SUB, MOV, and SHL are called mnemonics and these provide a way to instruct a machine. Sometimes these are also known as machine instructions. These instructions can be different for different microprocessor architecture, which means program written for one architecture is not compatible with another architecture.

Now, we present a program to print "Welcome to programming" in each of the four different programming languages—Pascal, C, Java, and Scala. This will give us some idea on how programming languages have evolved in the history of computer science.

Figure 1.2 shows code snippet for a Pascal program that prints "Welcome to programming" on the screen. The program starts with the keyword *PROGRAM*, which is followed by a program name; program name can be any valid identifier. The code that is responsible to print the message on the screen is placed between two keywords–*BEGIN* and *END*. In fact, *writeln* is a procedure that takes string,

```
PROGRAM sampleprogram;
BEGIN
  writeln('Welcome to programming');
END.
```

Figure 1.2: Sample Pascal Program

```
#include<stdio.h>
main() {
  printf("Welcome to programming");
}
```

Figure 1.3: Sample C Program

in this case, and outputs it to the screen, the output device. Please note that this procedure utilizes output device. Just to remind ourselves, the way to get handle to an output device is through the operating system, which runs this Pascal program. In case of high level programming languages, this method of getting handle of input and output devices is hidden from the programmers so that they can focus on solving computation problems at hand. There are many online resources available if you want to compile and run this program.

The program that can print "Welcome to programming" in C programming language is shown in Figure 1.3. There are many similarities between a Pascal program and a C program. Both contain a function, though named differently, to enclose statements; both use output function to print on the screen. But if we see the syntax, they look different. In C, we need to define a main function and put statements in between curly brackets. Also it uses *printf* instead of *writeln*. Further, if we are using a library function then we need to include that library file, normally a header file that contains that library function. In this program, the first line of code (LOC) does that. Once a header file is included all the functions within that header file are available to use in our program.

```
public class TestProgram {
  public static void main(String[] arguments) {
    System.out.println("Welcome to programming");
  }
}
```

Figure 1.4: Sample Java Program

```
object TestProgram {
  def main(arguments: Array[String]): Unit = {
    println("Welcome to programming")
  }
}
```

Figure 1.5: Sample Scala Program

Now, let's try to achieve the same using a Java program. In fact, Java is the closest language to Scala; Scala programs are compile to *.class* files. Figure 1.4 shows a Java program that prints "Welcome to programming" on the screen. Java syntax looks significantly different from C syntax. First of all, we need to create a class and then write a main method inside that class. We saw main function in a C program and we are seeing a main method in a Java program. There is some relation between two languages as Java language was influenced by C language. Java methods need access modifier, so do Java classes. We have *public* visibility for both the *TestProgram* class and the *main* method inside it. *println* is a method from *PrintStream* class and *System* class contains a field, *out*, of type *PrintStream*. We are invoking *println* method of *out* object and passing our message "Welcome to programming." Since the field *out* is static in *System* class, we can call it as *System.out*.

Finally, we now discuss our much awaited Scala program. Scala uses singleton object, a class with a single instance, in order to run the main method. So instead of class, we have object *TestProgram* that contains main method. There are syntactical differences between Java and Scala. Types are separated by a colon in Scala and come after identifiers. Java syntax requires types to be written before the identifiers. Also Scala uses new line as a separator between two statements or expressions. If we want to use two expressions in the same line then they should be separated by a semi colon.

Now, if we compare all four high level languages for which we wrote sample programs then we find *main* in three of them—C, Java, and Scala. In C, it is called main function. In Java and Scala, it is called main method. In Java, it is a static method; the concept is same in Scala as well with slightly different arrangements. Scala recommends singleton objects for static methods. Singleton objects are classes with only one instance; the instance is created the first time it is used, on demand.

All four languages have some pre-defined words like *class*, *object*, *PROGRAM*, *BEGIN*, *END*, etc. Also it is relatively faster to comprehend code written in Pascal, C, Java, and Scala compared to the code written in assembly language. That's why these languages are called high level languages; high in the sense that they are close to natural languages like English. If we see longer programs written in Pascal, C, Java, and Scala then we get a feeling that Scala is much more closer to a natural language. Similarly, Java is closer to natural language than C. So the language advancements are moving away from machine dependencies and trying to be as close as possible to human spoken languages.

1.4 Introduction to Scala

Scala was designed by professor Martin Odersky, who is also a co-designer of Java Generics. Also Dr. Odersky implemented a reference Java compiler. He is an academician who has significant industrial experience. This blend of experience has allowed him to create programming languages, which are intellectually challenging as well as of applied nature. He has done a great job of unifying object oriented paradigm with functional paradigm. If there is one programming language that brings functional programming seamlessly to large mass of industrial software engineers, that is Scala.

Scala programs run on a Java Virtual Machine (JVM). This has multiple advantages. One of the big advantages is JVMs are around for more than two decades and tools have matured. Also there are millions of applications running on JVMs. Since Scala programs compile to intermediate code, called byte code, developers don't have to modify their programs for new type of machines. If a new type of machines appear in the market, the machine manufacturer writes JVM for that machine and same Scala programs will run in the new machine architecture. This concept was pioneered by Java programming language, which is also credited for defining software engineering. Scala takes it one step further by seamlessly integrating functional programming with object oriented programming. Also Scala is a purely object oriented programming language. Further, recent advancements in Java were inspired by Scala.

With that short introduction, now let's write programs for the computational problems that we introduced in the beginning of this chapter. Figure 1.7 shows a Scala program that defines two numbers to be added, adds the numbers, and then prints the sum on the screen. The first line starting with *package* provides the package name; a Scala package is pretty much like a package in daily life in the sense that it is a name space to house classes, objects, traits, etc. that collectively achieve one or more computation goals.

Now, if we want to perform addition for another set, 13 and 29 that we presented earlier in this chapter, then we need to replace those two 1*s* by 13 and 29. The order does not matter as addition is commutative. Once we replace two operands

```
package com.equalinformation.scala.programs
object AddTwoIntegers {
     def main(arguments: Array[String]): Unit = {
          val firstNumber = 1
          val secondNumber = 1
          val sum = firstNumber + secondNumber
          println("The sum is "+sum)
     }
}
```

Figure 1.6: Program to Add Two Pre-defined Numbers

```
package com.equalinformation.scala.programs
import scala.io.StdIn._
object AddTwoIntegers {
     def main(arguments: Array[String]): Unit = {
          print("Enter the first integer: ")
          val firstNumber = readInt()
          print("Enter the second integer: ")
          val secondNumber = readInt()
          val sum = firstNumber + secondNumber
          println("The sum is "+sum)
     }
}
```

Figure 1.7: Program to Add Two Numbers

and re-run the program, we get the sum, 42, printed on the screen. This implementation is too specific as operands are hard coded; Every time we need to compute addition of two numbers, we have to replace those two operands. Now, we can improve this program by making it a bit more generic. In order to make it generic, we modify the program so that it takes two integers from the keyboard. So every time we need to compute addition, we just need to run the program and input two integers from the keyboard. In this way, we don't have to modify the program and re-compile.

Figure 1.7 presents the improved version. we use *readInt()* method to read an integer from the keyboard. Since this method is available in *StdIn* object, which is packaged in *scala.io* package, we have the import statement, *import scala.io.StdIn._*. The underscore, '_', at the end of the import directs compiler to import everything in the object *StdIn*. Note that we have re-used many LOCs from our previous program.

Re-usability is one of desirable attributes of computer programs; this is certainly true for Scala programs. Please note that while generalizing the program we did not write everything from scratch; we made least modification to meet modified requirements. The requirement change in this case was to read operands from keyboard, instead of hard coding in our program.

Now let's write a Scala program to solve second computational problem that we discussed in the beginning of this chapter. Figure 1.8 shows the implementation for data presented in Table 1. Ideally, data comes from some persistence systems. Since this volume is about Scala language exploration, we will make all the programs self-contained in terms of data.

Figure 1.8 implements an information base for the data presented in Table 1. Each category is first represented as a Map and the representation should be fairly obvious to read. We will discuss syntactical details later in other chapters. The program then creates a list that contains all the maps; this is to create a holistic data so that we can perform search by referring one data structure. The LOC that contains embedded *foreach* navigates all the values in the *taxonomyList*. We use pattern matching to find out the occurrence of *Felis*, which helps us to conclude the availability of taxonomy information for cat family. We will discuss pattern matching in Chapter 9.

1.5 Program Attributes

We saw several programs in earlier sections. When we embark for a professional programming career, there are certain attributes that fellow professionals would like to see in our programs. These program attributes not only make projects more successful but also help us foster our professional relationships, which create positive dynamics for the team as well as for the community of which we become part of. The following attributes are most common in industrial software engineering.

- Comprehensible
- Maintainable
- General
- Simple
- Modular
- Efficient

```
package com.equalinformation.scala.programs
object BioTaxonomy {
  def main(args: Array[String]): Unit = {
    val humanTaxonomy = Map("Kingdom" -> "Animalia",
                            "Phylum" -> "Chordata",
                            "Class" -> "Mammalia",
                            "Order" -> "Primates",
                            "Family" -> "Hominidae",
                            "Genus" -> "Homo",
                            "Species" -> "H. Sapiens")
    val dogTaxonomy = Map("Kingdom" -> "Animalia",
                          "Phylum" -> "Chordata",
                          "Class" -> "Mammalia",
                          "Order" -> "Carnivora",
                          "Family" -> "Canidae",
                          "Genus" -> "Canis",
                          "Species" -> "C. lupus" )
    val pigeonTaxonomy = Map("Kingdom" -> "Animalia",
                             "Phylum" -> "Chordata",
                             "Class" -> "Aves",
                             "Order" -> "Columbiformes",
                             "Family" -> "Columbidae",
                             "Genus" -> "Columba",
                             "Species" -> "C. livia")
    val catTaxonomy = Map("Kingdom" -> "Animalia",
                          "Phylum" -> "Chordata",
                          "Class" -> "Mammalia",
                          "Order" -> "Carnivora",
                          "Family" -> "Felidae",
                          "Genus" -> "Felis",
                          "Species" -> "F. catus")
    val taxonomyList = List(humanTaxonomy, dogTaxonomy,
     pigeonTaxonomy, catTaxonomy)
    var count = 0;
    taxonomyList.foreach(_.values.foreach(x => x match {
      case "Felis" => count += 1
      case _ => count += 0
    }))
    println("Total cat taxonomy found: "+count)
  }
}
```

Figure 1.8: Program for Biological Taxonomy Information Base

- Correct and accurate

Programs that are faster to read are said to be having better or higher *readability*. A program that we write today might look strange in about six months and hence it is important to write readable programs so that it is easy to maintain in the future. Generally, software applications last more than a year and it is also true that team members change over a period of time for various reasons. In this context, programs that are comprehensible are much more desirable. This is one of the reason why engineers and scientists came up with high level programming languages. High level programming languages allow us to write much more readable programs compared to assembly languages.

Maintainability of a program is closely related to readability of a program. A program that has higher readability is easier to maintain. We observed many development teams recommending a meaningful identifier instead of just a single letter. Also adding comments helps to understand the program better. Package naming, packages organization, configuration organization, etc. help to improve the maintenance of a code base.

In Section 1.4, we saw an example of making a program generic. Generic programs can cover more computational cases than a specific program. Of course, in some cases, programs have to be specific in order to provide better value. This decision has to be made based on who is buying the software. If the customers need specific programs then we deliver specific. But as a general principle, generic programs are more useful in the sense that they cover larger computational cases.

A computational problem can have multiple programming solutions. Simpler solutions are cost effective to maintain. In a typical software application life cycle, maintenance consumes more resources than original development. It is also our observation that people have different attitude toward simplicity in different dynamics. But for a long run, simplicity certainly pays off.

For a large and complex programs, it is very important to organize code so that navigation becomes faster. Also it is a common practice to group programs based on their functionality. In this way, if something has to be fixed for a particular functionality, then the developers know which module to look into. In a typical industrial software application, there can be hundreds of classes. Modularity can be implemented in different flavors. Writing several services and providing an interface to those services is an example of making code modular. Sometimes, modularity can also be achieved by packaging. In fact, closely related services are packaged together.

Efficiency is equally important. A program should be able to provide computational service by consuming minimum memory and minimum processing power.

When we design a program or collection of programs, it is important to keep efficiency in mind as there is no infinite computational power available. Even if an infinite computational power is available, we need to achieve the computation in a finite amount of time, relative to human life. So time is another factor to consider for efficiency.

Correctness indicates whether a solution satisfies requirements; generally, accuracy means precision. An accurate solution may not be correct. For example, we can get a solution that is accurate upto 5$^{\text{th}}$ decimal place with incorrect formula. On the other hand, we can right formula and wrong types and get unintentional rounding of decimal places. One of the things to keep in mind as a programmer is the selection of right data types and right programming constructs, which produce accurate results. Correctness might go beyond programming constructs including right algorithms and right formulae.

1.6 Conclusion

In this chapter, we discussed introductory computing concepts including widely used terminologies—computing and computation. Then we discussed tool aspect of computing by introducing computers components. In a typical modern digital computer, we find memory unit to hold run time information, processing unit to do processing, storage unit for persistence, input devices to provide input, and output devices to send output to. Every computer needs a software system to operate it, called operating system. We briefly discussed three most popular operating systesm—Linux, Mac OS, and Windows.

Next, we discussed programming languages. Programming languages can be categorized into two main categories—low level and high level. Machine code and assembly language are part of low level programming. Low level programming generally refers to programming using either machine code or mnemonics; mnemonics map to machine instruction sets. Scala, Java, C, and Pascal are categorized as high level programming; high level programming are closer to natural languages and use words from natural languages. Programs written in high level programming languages are much more easier to comprehend compared to programs written in assembly languages. Next, we introduced the topic of this book, Scala programming language. Then we provided solution to two computational problems discussed in the beginning of this chapter. Finally, we discussed elements of a good program. In Chapter 2, we will learn Scala fundamentals.

1.7 Review Questions

1. What is a difference between computation and computing?
2. List basic components of a typical modern digital computer.
3. What roles do operating systems play?
4. List three most popular operating systems today?
5. Why low level programming is not efficient in terms of program development?
6. Name at least five high level programming languages.
7. Write three differences between Scala and Java.
8. What is the philosophy behind Scala?
9. Is functional programming better than object-oriented programming?
10. What are some of the attributes of a good computer program?

1.8 Problems

1. Write a program to print "Scala is fun." on the screen.
2. Write a program to calculate the difference between two integers. *Pre-condition*: The two integers should be read from a keyboard. *Post-condition*: The difference should be displayed on the screen.
3. Write a program to print each letter of a string. *Pre-condition*: The string should be read from a keyboard. *Post-condition*: Each letter should be printed in a separate line.

1.9 Answers to Review Questions

1. According to Oxford English Dictionary, the word computation refers to mathematical calculation and computing refers to the use of computing machines in order to perform computation.
2. The basic components of a typical modern computers are:
   - Memory Unit

- Processing Unit
- Storage Unit
- Input Device
- Output Device

3. Operating systems administer the hardware. In other words, operating systems give life to the hardware. Also operating systems act as a mediator between hardware and application software.

4. The three most popular operating systems today are Linux, Mac OS, and Windows.

5. Low level programming languages require developers to program at register level. Further developers need to take care of fine grained details. Also the use of mnemonics is not as comprehensible as natural language based words.

6. Five high level programming languages are Scala, Java, Go, Swift, and Python.

7. The three differences between Scala and Java are:

   - Scala is a purely object oriented language but Java is not.
   - Scala was designed with the aim of combining object oriented paradigm with functional paradigm but Java was designed to be an object oriented language. Over a course of time, Java has adopted some functional programming features.
   - Scala had built in features for program parallelism from the beginning but Java introduced it much later.

8. Scala designer, Dr. Martin Odersky, believes that the object oriented paradigm can be combined with the functional paradigm to provide better computing experience. And has proven it.

9. It depends on the nature of computational problem. So there is no definite answer. For some problems, object oriented approach might be better, for other problems functional approach might work better. Further, there can problems that can benefit by the combination of these two programming paradigms.

10. Some of the attributes of a good computer program are:

    - Readable
    - General enough to cover wider computational problem space
    - Maintainable

- Simple
- Modular
- Efficient in term of memory and CPU

1.10 Solutions to Problems

1.
```
object SolutionToProblem1 {
  def main(arguments: Array[String]): Unit = {
    println("Scala is fun.")
  }
}
```

2.
```
import scala.io.StdIn._
object SolutionToProblem2 {
  def main(arguments: Array[String]): Unit = {
    print("Enter the first integer: ")
    val firstNumber = readInt()
    print("Enter the second integer: ")
    val secondNumber = readInt()
    val difference = firstNumber - secondNumber
    println("The difference is "+difference)
  }
}
```

3.
```
object SolutionToProblem3 {
  def main(arguments: Array[String]): Unit = {
    print("Please enter a string: ")
    val inputString = scala.io.StdIn.readLine()
    inputString.toString.foreach(println)
  }
}
```

Chapter 2

Scala Fundamentals

(content here)

2.1 Literals

(content here)

2.2 Identifiers and Keywords

(content here)

2.3 Types

(content here)

2.4 Declarations and Definitions

(content here)

2.5 Expressions

(content here)

2.6 Conclusion

(content here)

2.7 Review Questions

(content here)

2.8 Problems

(content here)

2.9 Answers to Review Questions

(content here)

2.10 Solutions to Problems

(content here)

Chapter 3

Classes and Objects

(content here)

3.1 Class Members

(content here)

3.2 Class Definition

(content here)

3.3 Object Definitions

(content here)

3.4 Conclusion

(content here)

3.5 Review Questions

(content here)

3.6 Problems

(content here)

3.7 Answers to Review Questions

(content here)

3.8 Solutions to Problems

Chapter 4

Control Structures

(content here)

4.1 For Expressions

(content here)

4.2 While Loops

(content here)

4.3 If Expressions

(content here)

4.4 Exception Handling

(content here)

4.5 Conclusion

(content here)

4.6 Review Questions

(content here)

4.7 Problems

(content here)

4.8 Answers to Review Questions

(content here)

4.9 Solutions to Problems

Chapter 5

Operators

(content here)

5.1 Operators as Methods

(content here)

5.2 Arithmetic Operators

(content here)

5.3 Relational and Logical Operators

(content here)

5.4 Bitwise Operators

(content here)

5.5 Operator Precedence and Associativity

(content here)

5.6 Conclusion

(content here)

5.7 Review Questions

(content here)

5.8 Problems

(content here)

5.9 Answers to Review Questions

(content here)

5.10 Solutions to Problems

Chapter 6

Data Input and Output

(content here)

6.1 Single Character Input

(content here)

6.2 Single Character Output

(content here)

6.3 Reading From a File

(content here)

6.4 Writing to a File

(content here)

6.5 Navigating Directories

(content here)

6.6 Conclusion

(content here)

6.7 Review Questions

(content here)

6.8 Problems

(content here)

6.9 Answers to Review Questions

(content here)

6.10 Solutions to Problems

Chapter 7

Traits

(content here)

7.1 Traits as Interfaces

(content here)

7.2 Construction Order

(content here)

7.3 Trait Members

(content here)

7.4 Multiple Inheritance

(content here)

7.5 Traits with Implementations

(content here)

7.6 Conclusion

(content here)

7.7 Review Questions

(content here)

7.8 Problems

(content here)

7.9 Answers to Review Questions

(content here)

7.10 Solutions to Problems

Chapter 8

Functions

(content here)

8.1 Functions as Methods

(content here)

8.2 Anonymous Functions

(content here)

8.3 Functions as Values

(content here) sectionFunction Parameters (content here)

8.4 Higher-Order Functions

(content here)

8.5 Closures

(content here)

8.6 Currying

(content here)

8.7 Conclusion

(content here)

8.8 Review Questions

(content here)

8.9 Problems

(content here)

8.10 Answers to Review Questions

(content here)

8.11 Solutions to Problems

Chapter 9

Pattern Matching

(content here)

9.1 Case Classes

(content here)

9.2 Variable Patterns

(content here)

9.3 Typed Patterns

(content here)

9.4 Pattern Binders

(content here)

9.5 Literal Patterns

(content here)

9.6 Stable Identifier Patterns

(content here)

9.7 Constructor Patterns

(content here)

9.8 Tuple Patterns

(content here)

9.9 Extractor Patterns

(content here)

9.10 Sequence Patterns

(content here)

9.11 Infix Operation Patterns

(content here)

9.12 XML Patterns

(content here)

9.13 Regular Expression Patterns

(content here)

9.14 Irrefutable Patterns

(content here)

9.15 Type Patterns

(content here)

9.16 Conclusion

(content here)

9.17 Review Questions

(content here)

9.18 Problems

(content here)

9.19 Answers to Review Questions

(content here)

9.20 Solutions to Problems

Chapter 10

Inheritance and Composition

(content here)

10.1 Extending Classes

(content here)

10.2 Overriding Methods and Fields

(content here)

10.3 Abstract Classes

(content here)

10.4 Invoking Superclass Constructors

(content here)

10.5 Polymorphism and Dynamic Binding

(content here)

10.6 Composition

(content here)

10.7 Conclusion

(content here)

10.8 Review Questions

(content here)

10.9 Problems

(content here)

10.10 Answers to Review Questions

(content here)

10.11 Solutions to Problems

Chapter 11

List Processing

(content here)

11.1 List Construction

(contenthere)

11.2 Operations

(content here)

11.3 Patterns

(content here)

11.4 List Class

(content here)

11.5 List Ojbect

(content here)

11.6 Conclusion

(content here)

11.7 Review Questions

(content here)

11.8 Problems

(content here)

11.9 Answers to Review Questions

(content here)

11.10 Solutions to Problems

Chapter 12

The Scala Collections Framework

(content here)

12.1 Mutable versus Immutable Collections

(content here)

12.2 Sets

(content here)

12.3 Maps

(content here)

12.4 Sequences

(content here)

12.5 Tuples

(content here)

12.6 Conclusion

(content here)

12.7 Review Questions

(content here)

12.8 Problems

(content here)

12.9 Answers to Review Questions

(content here)

12.10 Solutions to Problems

Chapter 13

Actors

(content here)

13.1 The Components of Actors

(content here)

13.2 Creating Actors

(content here)

13.3 Sending and Receiving Messages

(content here)

13.4 Life Cycle

(content here)

13.5 Channels

(content here)

13.6 Linking

(content here)

13.7 Conclusion

(content here)

13.8 Review Questions

(content here)

13.9 Problems

(content here)

13.10 Answers to Review Questions

(content here)

13.11 Solutions to Problems

Chapter 14

XML Processing

(content here)

14.1 XML Literals

(content here)

14.2 Serialization and Deserializing

(content here)

14.3 Data Extraction

(content here)

14.4 Pattern Matching

(content here)

14.5 Loading and Saving

(content here)

14.6 Conclusion

(content here)

14.7 Review Questions

(content here)

14.8 Problems

(content here)

14.9 Answers to Review Questions

(content here)

14.10 Solutions to Problems

Chapter 15

Parsing

(content here)

15.1 Lexical Analysis and Parsing

(content here)

15.2 Running Parser

(cotent here)

15.3 Regular Expression Parser

(content here)

15.4 JSON Parser

(content here)

15.5 Error Handling

(content here)

15.6 Conclusion

(content here)

15.7 Review Questions

(content here)

15.8 Problems

(content here)

15.9 Answers to Review Questions

(content here)

15.10 Solutions to Problems

Chapter 16

GUI Programming

(content here)

16.1 Simple Application

(content here)

16.2 Events

(content here)

16.3 Panels

(content here)

16.4 Layouts

(content here)

16.5 Example Application

(content here)

16.6 Conclusion

(content here)

16.7 Review Questions

(content here)

16.8 Problems

(content here)

16.9 Answers to Review Questions

(content here)

16.10 Solutions to Problems

Chapter 17

Unit Testing

(content here)

17.1 Unit Testing in Scala

(content here)

17.2 ScalaTest

(content here)

17.3 ScalaCheck

(content here)

17.4 JUnit

(content here)

17.5 TestNG

(content here)

17.6 Tests as Specifications

(content here)

17.7 Conclusion

(content here)

17.8 Review Questions

(content here)

17.9 Problems

(content here)

17.10 Answers to Review Questions

(content here)

17.11 Solutions to Problems

Bibliography

[OLC17] One laptop per child. http://one.laptop.org/, 2017.

[UNL13] United nations adult literacy rate. http://data.un.org/Data.aspx?d=SOWC&f=inID