Programming Language Semantics: Reminder and basic notions

Tarek Boutefara – t_boutefara@univ-jijel.dz – Version 0.1, 01-10-2024

Table of Contents

1. Introduction

Before embarking on formal approaches to proving programs, it is important to review a number of basic concepts, such as the notion of software, programs and tests.

2. Software

Software can be defined as: 'the programs and other operating information used by a computer'. (Oxford dictionary)

The word 'Software' is be defined in comparison with 'Hardware', which constitutes the physical part of the computer system. Software allows the user to manipulate and operate the system.

Softwares can be classified according to various criteria. The most commonly used criterion is its purpose. According to this classification, there are two main categories:

- Operating systems: used to operate the machine (the computer), they enable resources to be used and provide basic functions such as access to files, use of peripherals and devices, and control and sharing of resources between the various programs running.
- Applications: these are used to carry out specific tasks for which they have been designed. A distinction is made between :
  - Drivers: enable the operating system and other applications to communicate with peripherals.

- Utilities: used to perform diagnostic, configuration or optimisation tasks.
- Applications: can be custom or standard. Applications enable specialised tasks to be carried out, such as office tasks, image processing, video games and web browsers.

Software is created using a Software Engineering process depending on the software to be developed. The choice varies from a simple cascade process (which is generally used for systems with stable requirements) to more complicated processes that allow iterability and adaptability according to the different constraints that may arise during the project.

The complexity that software carries with it, given its nature and the process by which it is produced, leads us to reflect on the possibility of testing and proving software. Indeed, being able to test and prove that the software performs the task for which it is designed in a correct manner is an essential element in ensuring its quality and preventing data loss during its use. Bugs can also have catastrophic consequences if the data affected is sensitive (e.g. authentication and financial information).

3. Test and debugging

To ensure the quality of the software, it is important to 'test' it before deploying it. The testing phase is an essential part of all Software Engineering processes, from the very beginning of the discipline (waterfall development).

We can distinguish between testing the software itself and testing its components. Component testing comes first and aims to assess the quality of the components and their integration. Software testing, on the other hand, aims to determine whether the software complies with and satisfies the specification developed (the functional requirements defined during the analysis phase).

3.1. Unit tests

Unit testing evaluates each component individually. The aim is to ensure that the component under test conforms to the detailed design defined for each component. This test is generally carried out using a test data set. The data set must define the input data and the results expected after execution using this data. This testing approach is known as dynamic analysis.

An error is considered to exist if the component under test returns values other than those expected as results from the data used for the test. In this case, the component is sent back to the previous phase (development phase) to be reworked (rewriting, rectification or debugging).

Example:

Consider the following class:

JAVA

```
public class Exemple {
    public static int somme(int a, int b){
        return a + b;
    }
}
```

The unit test using the JUnit library could be as follow:

JAVA

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class JUnitMethodTest {

    // L'annotation précise qu'il s'agit d'une méthode de test
    @Test
    public void somme(){
        try {
            log.info("Starting execution of somme");

            // Jeu de données de test
            int expectedValue=20;
            int a=15;
            int b=5;

            // Calcul du résultat
            int actualValue=Exemple.somme(a, b);

            // Vérification que le résultat égale la valeur prédie
            assertEquals(expectedValue, actualValue);

        } catch (Exception exception) {
            exception.printStackTrace();
        }
    }
}
```

3.2. Integration test

The integration test comes after all the unit tests have been passed and the components have been assembled. The test is conducted using an approach similar to that used for unit testing. During this phase, the software is built progressively while testing the new configurations resulting from the assembly actions.

At the end of the test, the resulting files are saved, and a report is produced.

3.3. Installation (reception) test

Once the software has been built (assembly of the software components), it is time to install it in its operating environment (known as the ‘production’ environment). The aim of the installation test is to ensure that the software behaves in accordance with the specification defined during the analysis phase.

A negative test result can vary from a minor correction of bugs detected during the test to a complete rejection of the software if it does not meet the customer’s needs and expectations (serious error such as incomplete data, exception case not handled, and many others). Successful completion of this test confirms that the customer has received the software (hence the name ‘acceptance test’).

3.4. In summary

There are essentially three types of test: unit tests, integration tests and installation tests. The tests are run in this order. Unit tests come first to confirm that each component works correctly. The integration test validates the assembly of the components. The acceptance test validates the software and confirms that it has been received by the customer.

These three types of test are based on :

- Data sets prepared for the test: this is a set of pairs (input data, expected result). The input data is passed to the component (simple or compound) to obtain the calculated value. The calculated value is compared with the expected value (dynamic analysis).
- Test run by the customer in concrete cases.

In both cases, we can observe the informal nature of these tests. Indeed, the quality of the unit tests depends mainly on the quality of the data set and its completeness. As a result, it is impossible to imagine a test that contains all the possible inputs and all the expected results (in which case, the development of the software itself becomes questionable).

Installation tests include also an unformal dimension. These tests are based on the required specification and the documentation provided with the software. In both cases, the descriptions are informal and generally written in natural language.

It is therefore important to provide formal methods for verifying the quality of a piece of software (or a component). We refer to this as 'program proof' or static analysis.

4. Program proof

The proof of a program is the certification of the correctness of a program using semi-automatic tools. The proof of a program can also be used to formalise the specification of a program.

This proof is generally in the form of property verification. In this context, it is possible to define two types of property:

- Syntactic properties: concern the source code of the program, such as :
  - The program (source code) contains a loop,
  - The programme (source code) consists of 'n' functions.
- Semantic properties: concern the behaviour of the program, such as :
  - The program does not cause exceptions,
  - Program X is equivalent to program Y,
  - The program is correct in relation to its specification.

Rice's theorem: Any non-trivial extensional property relating to programs is undecidable.

From this theorem, we can conclude that it is difficult to verify all possible non-trivial semantic properties. So we have to make do with an approximate analysis of the programs and check only certain properties.

5. Notion of programme

In this reminder, we have referred to software as a set of programs. We have also mentioned program proof as an alternative method to dynamic analysis, which relies on a set of data to test a program. So, it is important to recall the notions of program.

5.1. Program

In computing, a program is a list of instructions for executing a task on a machine, written in a conventional form.

5.2. Instruction

An instruction is a command that tells the machine what to do. Successive execution of instructions enables complex behaviour to be obtained and sophisticated processing of information to be carried out.

5.3. Orgines: Turing's machine and von Neumann's architecture

The notions of instruction and program used today as basic elements of programming (essentially procedural and object-oriented) derive directly from the Turing machine.

5.4. Turing machine

Alan Turing was one of the mathematicians who worked on the notion of 'computability'. Calculability is a field of mathematics that seeks to identify the class of functions that can be calculated using an algorithm. We say that a function f is calculable if there is a precise method that can be used to calculate the image f(x) given x.

Alan Turing proposed a machine (known as the Turing machine) to represent all these functions. It was later demonstrated (by Church) that any calculable function is a function that can be written as a program on the Turing machine.

A Tuting machine consists of :

- A set of symbols.
- A ribbon of unlimited length subdivided into identical cells. Each box can contain a symbol.
- A read/write head. This head points to a single square at a time, can read and write in the square pointed to, and can move left and right along the ribbon.

An example of a Turing machine.

The Turing machine defined on the alphabet (the set of symbols) {0, 1} and governed by the following instructions:

- Write 1 if the square pointed to contains 0,
- Write 0 if the square pointed to contains 1,
- Move right to point to the square just to the right of the square pointed to,
- Move left to point to the square just to the left of the square pointed to.
- Stop.

Each Turing machine has a set of states q_i. For each state, there are two instructions to be executed according to the symbol at the box pointed to. So each instruction is a quadruplet $\{q, a_i, b_i, q'\}$ where :

- q and q' are two states,
- a_i is a symbol,
- b_i is an instruction (of the five defined above).

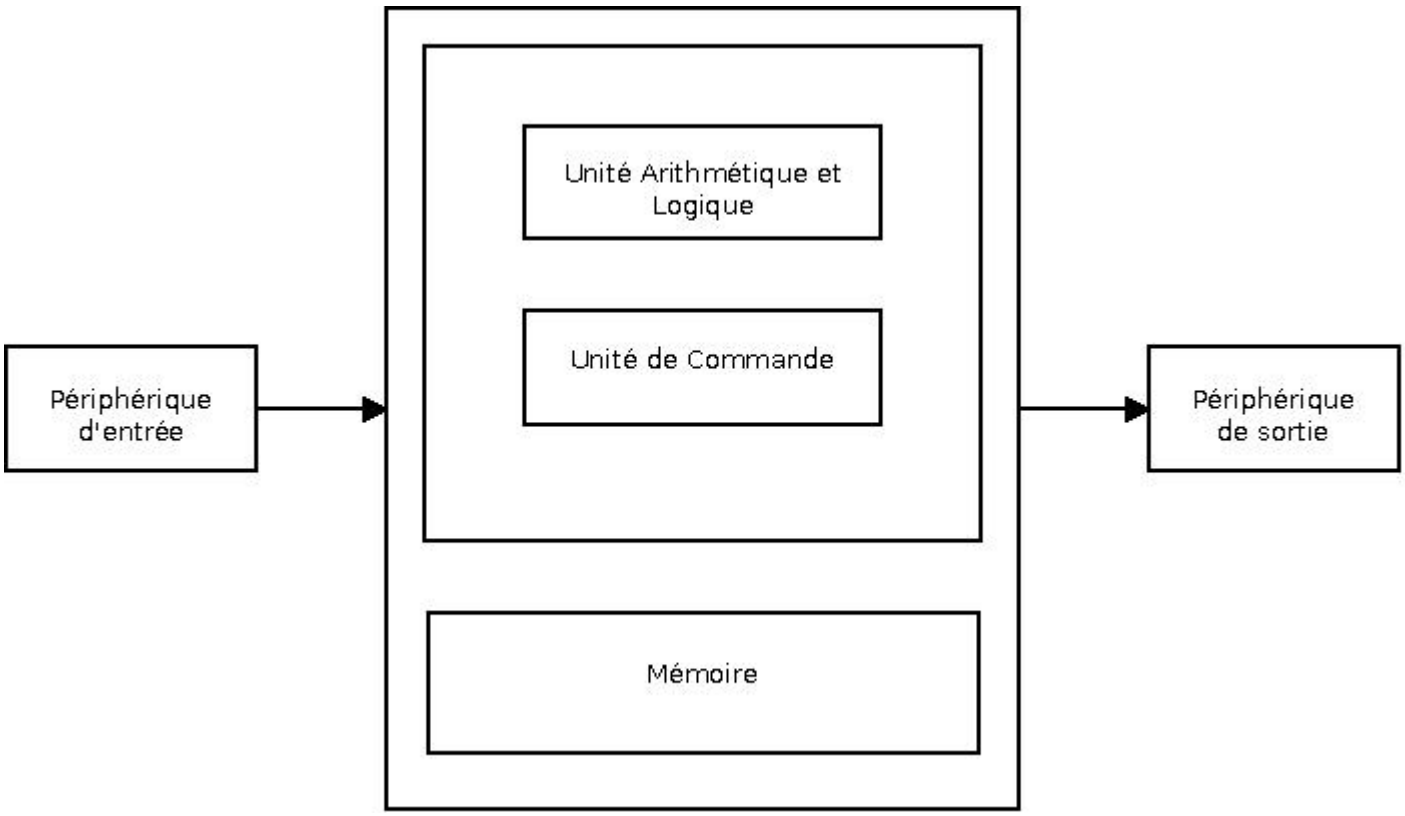
5.5. Von Neumann architecture

The Von Neumann architecture describes a physical machine that implements the theoretical Turing machine. This architecture is the model on which current computers are based. The essential feature of the Von Neumann machine is procedurality. This means that any problem must be described as a sequence of operations.

A computer based on this architecture consists of a memory and a processor. The memory plays the role of the ribbon: it contains the data. It also contains the programme to be executed, since it is binary data. The processor extracts the program instructions one by one, interprets them and executes them until the problem is solved. During execution, the data is transformed into a succession of states, the last state being the result.

The processor is made up of two units:

- Control unit: This part is responsible for scheduling the various stages in the processing of an instruction. To do this, the processor needs its own memory, independent of the main memory. This memory is made up of a number of registers which allow very fast access to store the results of operations. Registers C.O. and R.I. are special registers. The first is the program counter, which does not store data but points to the next instruction to be executed. The second is the (instruction register) which contains the instruction to be executed.
- It operates on the general registers as well as on the temporary registers (invisible to the programs and used for calculations).



These elements are necessary for the implementation of simple instructions that make it possible to create a Turing machine. Indeed, if we look at the Turing machine, which offers an infinite memory space (the ribbon) and a space that stores the program (with no particular form apart from the structure of the instruction). However, the Turing machine does not define a structure that interprets the instruction, just as it does not define a structure that executes the instruction (checking the contents of the box pointed to, for example).

5.6. Summary

The Turing machine has made it possible to define the notion of computability. We now know that any computable function can be written as a program on the Turing machine. The Von Neumann architecture is a concrete expression of the Turing machine. Therefore, we can easily prove that any program on the Turing machine can be written as a program on a machine based on the Von Neumann architecture, and by extension, any computable function can be written as a

program on a machine based on the Von Neumann architecture.

From this, we can say that it is possible to develop proof methods based on the semantics of instructions (static analysis) while respecting the origin and definition of the notion of instruction, which forms the basis of programs, which in turn form the basis of software.

6. Summary of the first chapter

In this first chapter, we have recalled the basic notions linked to the notion of software and necessary to assimilate the notion of proof (and particularly formal proof) of a program.

We have seen how, on the basis of the notion of computability, it was possible to construct a theoretical machine (Turing's machine) and then propose a physical machine that can execute programs that compute functions.

Although procedurality is the main characteristic of programs written for a Von Neumann architecture, the equivalence of the Turing machine to other formal systems has enabled other programming paradigms to be developed that do not rely on procedurality.

For example, the equivalence between the Turing machine and Kleene's recursive functions has made it possible to write recursive functions (defining a function that 'calls itself').

In the next chapter, we will look at the equivalence between the Turing machine and the λ-Calculus. This equivalence gave rise to the functional programming paradigm. We will look at the basics of the λ-Calculus, how to translate them into functional programs and how to prove them.

Version 0.1
Last updated 2024-10-13 00:39:20 +0100