Why is programming difficult?

Attila Egri-Nagy

v2018.03.10

It is often debated whether programming is difficult or not. However, at some point every developer hits a wall, when writing software becomes challenging. Where does the difficulty come from? This can be answered simply: in order to program a computer, one has to be a computer.

" To understand a program you must become both the machine and the program. "

Alan J. Perlis. Special feature: Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13, 1982

Programming is the act of creating computer programs. On the surface it looks like just writing weird looking text, obeying strict rules. However, the point is that the written source code generates processes in computing devices. The outcome of such a computational process is either a desirable piece of information (output), or the dynamics itself is a required behaviour (interactive applications, servers).

Programming is just one activity in software engineering, which is a lot bigger endeavour. To start with, it is in a social context. A product is created by people for users, who are also human beings. Consequently, its philosophy is more involved. If we need to single out a piece of wisdom, 'Be nice to each other' might be the most important maxim to follow. Another source of problems is the complexity of the systems we are building.

When programming, what are we doing exactly? First, we need to **imagine a computational process** we want to create. Second, we have to **code** that into a particular language. Alternatively, in exploratory programming, we write the code first, then after seeing the result, we try to imagine the process.

The coding part is easier, since programming languages are smaller and by design simpler than natural languages. It is way quicker to learn a programming language than a foreign language. Therefore, the difficulty must be in the imagination part.

Coding and programming are often used as synonyms. Here we view coding just as a stage of programming.

Imagining computation means playing out a symbolic mechanism in our head. Our brain is the runtime, we execute the code in our thinking. When writing assembly code, we envision bit-flips in registers and track data moving between memory and the processor. In object-oriented programming we have to picture the interaction patterns in the relation network of objects. In functional programming we have to visualize the structure of function calls and the substitution of arguments.

We emulate the computer by thinking about the running code. In other words, there is a morphic relation between the computation happening in the computer and our thoughts. By emulation we mean a strict one-to-one correspondence, or to use the technical term, an *isomorphism*. Today, almost any computer can recreate the behaviour of 8-bit computers from the 80's. That is not surprising at all. A more powerful machine can emulate a less powerful one, simply by having more memory and faster processors to make up for any difference between the machines. Doing the other way around, trying to emulate a computer with less resources, difficulties arise. This is what

Which computer exactly? This is a nontrivial question, which turns out to be a crucial one in computing education research. It is called 'notional machine', defined as "an abstract computer responsible for executing programs of a particular kind."

Juha Sorva. Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13(2):8:1–8:31, 2013 happens when we program.

The brain is a very lousy digital computer. It cannot focus strictly on a single data item. It is more like an association machine, whatever you think of, many other thoughts come into play. It also has severe limitations in scalability. The great thing about computers that they do things very fast and process a huge number of data items. We can keep only a small number things in our head at a time. We can imagine small cases, but errors might pop up for bigger instances of problems. We are also not too good in exploring all combinatorial possibilities, we tend to check the familiar cases. That's why we need to generate our test cases. The brain has vast resources, but of different type, and often biased evolutionary and culturally.

The history of programming language development can be viewed as a sustained effort to avoid this difficulty. Language features are there to reduce the cognitive load. If the machinery is hidden, we don't need to imagine it. This of course just pushes the semantics of the program to a higher level mechanism.

A historical analysis of the development of programming through the corresponding notional machines might give us useful insights. It hints the possibility of a cognitive comparison of language difficulty. Given a reliable complexity measure of notional machine definitions (if such a thing possible), might benchmark programming paradigms for education – with the danger of possibly fuelling flame wars.

Languages with interactive REPLs (read-eval-print-loop) also ease the cognitive load. Expressions can be evaluated separately, giving convenient access to the stages of the computational process. Still, there is a need to put these parts together and imagine the unfolding computation. **Tracing, the mental execution of source code remains to be a requirement for programming.**