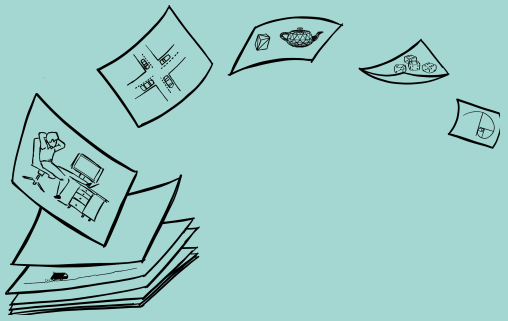


2^5 Things I Learned in Computer Science

University of Basel, Dept of Mathematics and Computer Science

2^5 Things I Learned in Computer Science

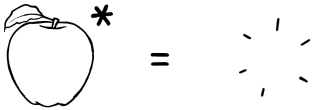
Author’s Note

Studying Computer Science is like taking a deep dive into an endless ocean of ideas, concepts, and tools. It is easy to get lost in this wide landscape, even more so if it is not one’s main field of expertise.

This is why Computer Science students of the University of Basel set out to identify the essential concepts and ideas they discovered during their studies. The intention was to go beyond the scope of traditional textbooks and also include the pecularities that are not formally part of typical Computer Science curricula. Finally, the goal was to present the fruit of this work to people outside of our field. The project was organized in the form of a Master’s level seminar, which had its first iteration during the spring semester 2019.

The result is **Things I Learned in Computer Science**, a collection of short illustrated articles. It is available in the form of this small booklet, and online on the companion website `https://tilics.dmi.unibas.ch`

Have fun discovering!



or

or

or

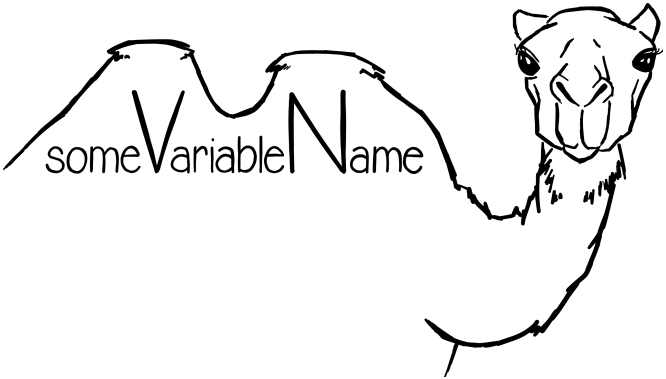
...

Kleene's Star is not Blah Blah

Computers are great at repetitive tasks. The most beautiful and concise way of expressing repetition is Kleene's star, namely the symbol $*$ which stands for **zero or more repetitions**. In Computer Science theory, the letter A followed by the Kleene Star, $\texttt{A}^*$, either means

- `""` (the word of length 0), or
- `"A"` (the word consisting of one A), or
- `"AA"` (the word consisting of exactly two As), etc

A related but different concept is a *wildcard* where the star is a placeholder (instead of a sign of repetition). For example `*.txt` selects all files that end in "txt". Kleene's star is not common in daily life, but wildcards are: A prominent example is "blah blah".


someVariableName

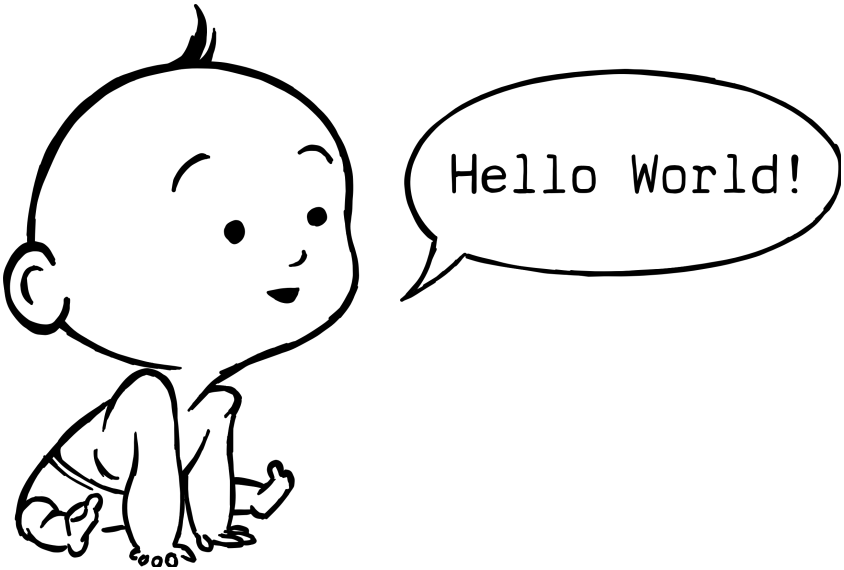
1

CamelCase, or why coding conventions matter

One problem that often arises when writing code is the naming of variables. Often multi-word names are used for variables but in most programming languages it is not possible to use spaces in a variable name.

One programming style that tries to structure this is called CamelCase. It is the practice of writing each word in the middle of a phrase with a Capital letter. This greatly improves readability.

```
thisGreatlyImprovesReadabilityComparedTo
    whenwewouldusenocapitalizationatall
```

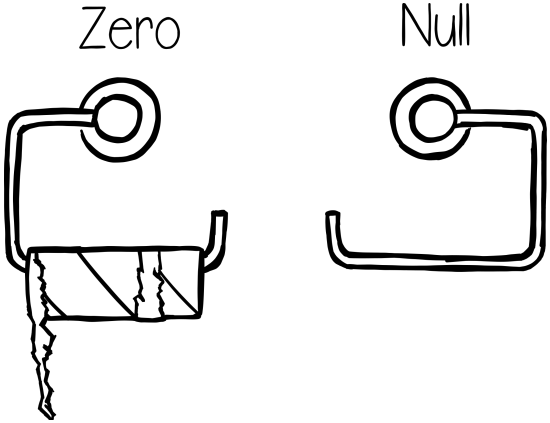

Hello World!

2

The first words of a computer scientist

The first piece of code you usually find in a programming book or tutorial is a “Hello World!” program. Its purpose is to display the text `Hello World!`.

Since it is rather easy to write a program to print text, one can familiarize themselves with the basics of a programming in a certain language. Due to this tradition, these are usually the first lines of code a programmer writes in their life.

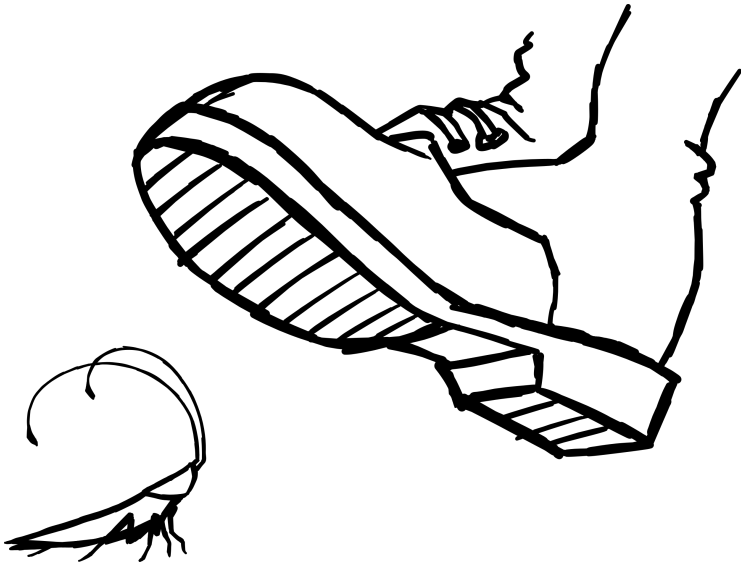

Zero
Null

3

Zero is not null

The `null` value has a special meaning in computer science. It is used to declare that a value is not yet defined or known.

The speciality of `null` is that although it shows us that a value is missing, it is itself a value. So it is possible to compare two non existing values. Suppose we have data about a patient; it does make a different if the patient has no disease (`0`) or is not yet diagnosed (`null`).



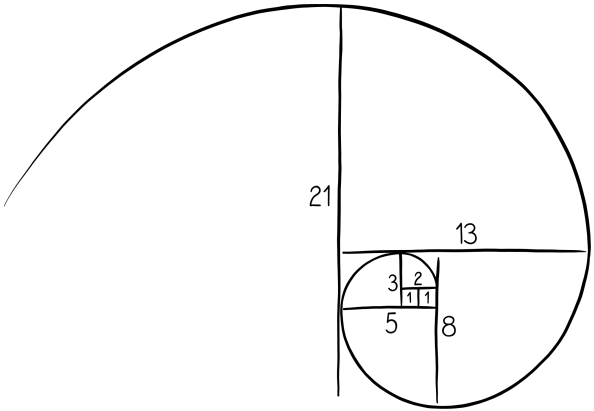
4

We all hate Bugs

Bugs creep into your house without notice, there they eat your food and clog your sink. They hide in corners and behind large furniture and are most of the times hard to find. If you find a bug that was bothering you for a long time the feeling of success is rewarding.

A bug in software is very simmilar. It’s usually a behavior of the software which is not intended. It is often very hard to find bugs in software and can sometimes be attributed to one single character that was wrongly placed.

The act of finding bugs is called Debugging. Like pest control, the code is searched and cleaned from every bug that influences the behavior negatively. Unfortunately, while fixing software bugs the programmer accidentally adds other bugs. The cicle is endless.



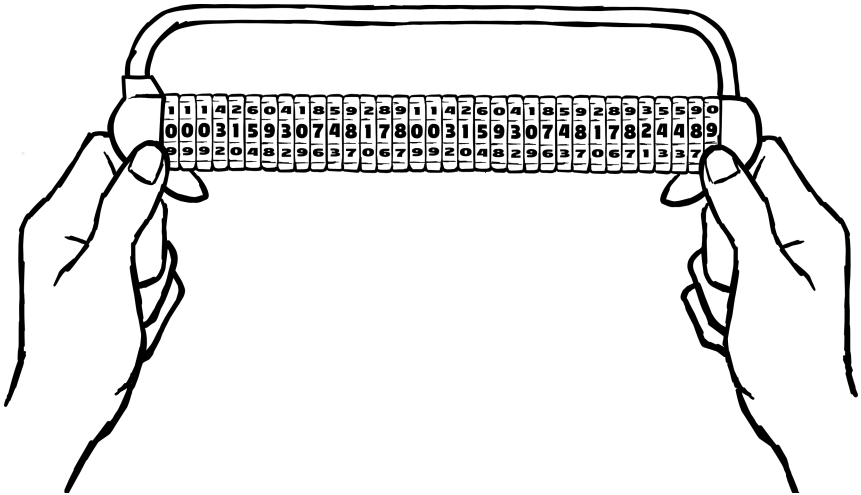
10th Fibonacci Number: 13 + 21 = 34

Sometimes a problem can be solved by solving the same problem, but smaller

5

Imagine a problem that you can only solve if you know the solution to smaller instances of the same problem. Computing, for example, the 10th element from the famous *Fibonacci sequence*, will require you to sum up the 9th and 8th Fibonacci number. But if you don’t know the 10th number, you will most certainly not know the 9th and 8th either. So you‘ll have to go deeper: In order to obtain the 9th number, you need to add the 8th and 7th element. Knowing the 8th will require the 7th and 6th. And so on. Doing this is called *Recursion*.

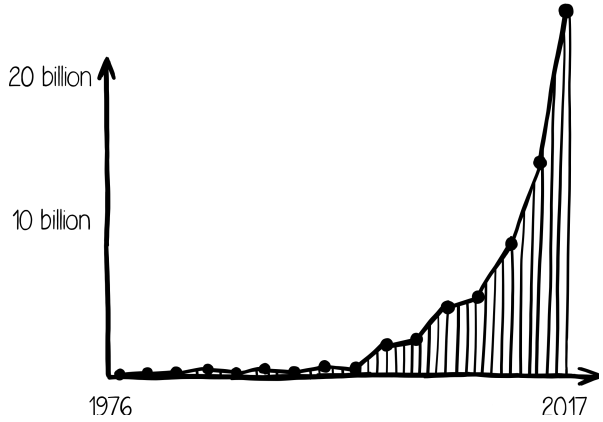
But when to stop? Recursion only works if there are certain smallest problems for which you *know* the solution, otherwise you’ll keep fragmenting forever. Continuing with the approach above will eventually lead you to computing the 1st and 2nd Fibonacci number, which are defined to be 0 and 1, respectively. Knowing these will allow you to hand the results all the way to the top and finally obtain the desired 10th Fibonacci number.



"Brute Force" means you are weak

For some problems in Computer Science, it can be proven that there are no “efficient” ways of solving them. In this case, the only resort is **brute force**. Brute force means that you let the computer try out every possible –and we really mean *every* single, possible and potential– value that solves the problem, until you succeed.

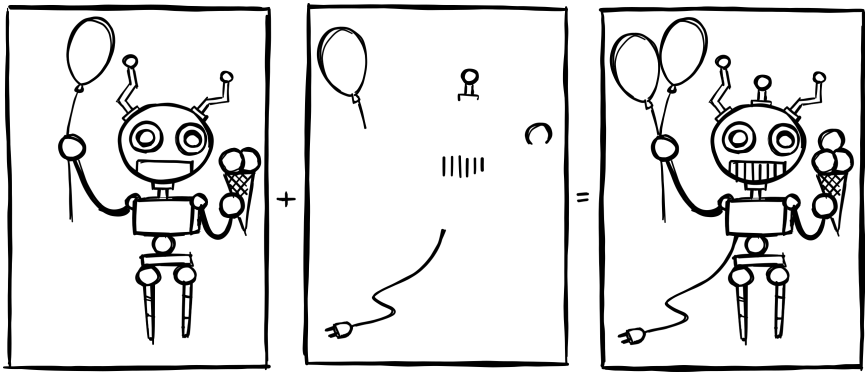
Several cryptographic algorithms are based on such problems where the secret key can only be guessed or found by exhaustively trying out all possible values during a few billion years. Confronted with such time scales, attackers find themselves in a very weak position.


Transistors
20 billion
10 billion
1976
2017

Moore's Law

In 1965, the technological progress lead Gordon E. Moore to formulate his rule of thumb that became famous under the name "Moore's Law". It states that the production costs of microchips used in electronic circuits would fall, leading to smaller and more powerful microchips at a lower overall cost. He estimated that the number of components on a single chip to double roughly every 12 months.

This law has been updated several times over the following decades. First the increase in the number of transistors used in a single chip was the driving force behind the exponential progress. Since the mid-1990s, and until the early 2000s, the continuation of the trend was mostly due to the downscaling of the components themselves. Today, as the manufacturing process is hitting physical limits, there is a development towards integrating functions, that until now were provided by separate components, into a single chip.



8

Why Patching is like Picture Puzzles

If you have a buggy version of a program and a corrected one: why send the whole program instead of just the small differences?

This is how updates for your smartphone work: you will receive the set of differences –a so called *patch*– that your smartphone applies to the old and buggy program.

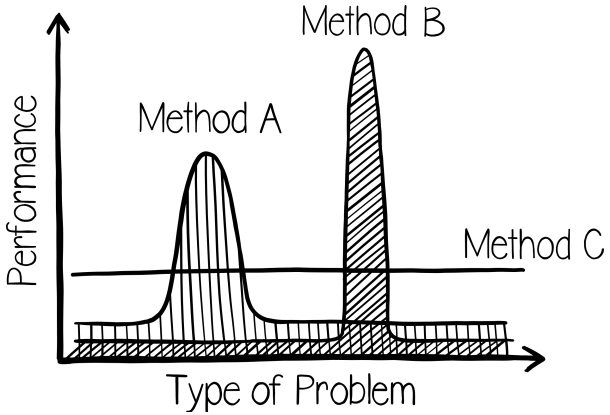
Now go and find the differences: Did we spot them all?

Being lazy is a good thing

A lazy person is always searching for ways to make things faster and more efficient.

Instead of doing the same thing over and over again, a good computer scientist tries to automate the task. This not only improves the reliability, but also saves time in the long run. Furthermore, a good computer scientist is also lazy when it comes to writing code. Rather than solving a specific problem, a good programmer develops a generic solution that can be reused for similar problems.

Good programmers are diligent in order to be lazy. Or with other words: They are strategically lazy. And that’s a good thing.

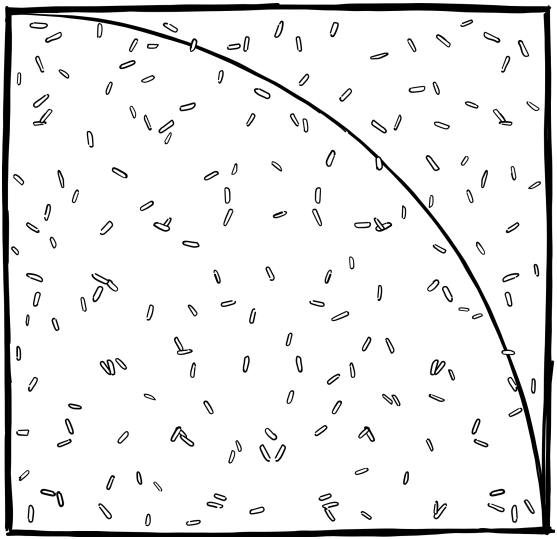

Method B
Method A
Performance
Method C
Type of Problem

There is no such thing as a free lunch

10

In Computer Science we have a wide range of very different problems and accordingly many approaches to tackle them. We might for example try to find some ‘optimal’ solution for a search problem, but it is unclear which specific algorithm one should use. So it is only natural to think about algorithms that always provide us the best solution, regardless of the problems subtleties.

However, the ‘No Free Lunch’ theorems state, that no single method works better than any other for all possible problems. Instead, it is necessary to select the method based on the problem (or data) at hand. That is, there is always a cost associated with selecting a method and unfortunately there is no such thing as a free lunch.

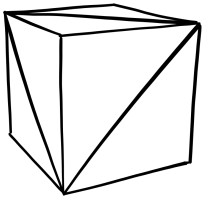


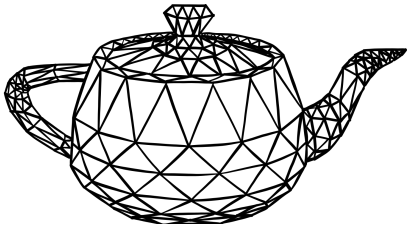
How to calculate Pi with rice

Draw a square and a quatercircle in it on a paper and throw a hand full of rice on it. Done!

Well almost done. Now you count how many grains of rice landed inside the square and how many landed inside the quatercircle. The ratio of these two numbers is an approximation of the ratio of the two areas Pi*r$^{2/r}$2=Pi/4. Multiply the ratio by 4 and you get Pi, well an approximation. If you are not happy with the accuracy just use more rice.

This is one example for the Monte Carlo method. Key is to avoid computing a complicated formula by using random samples in a clever way.

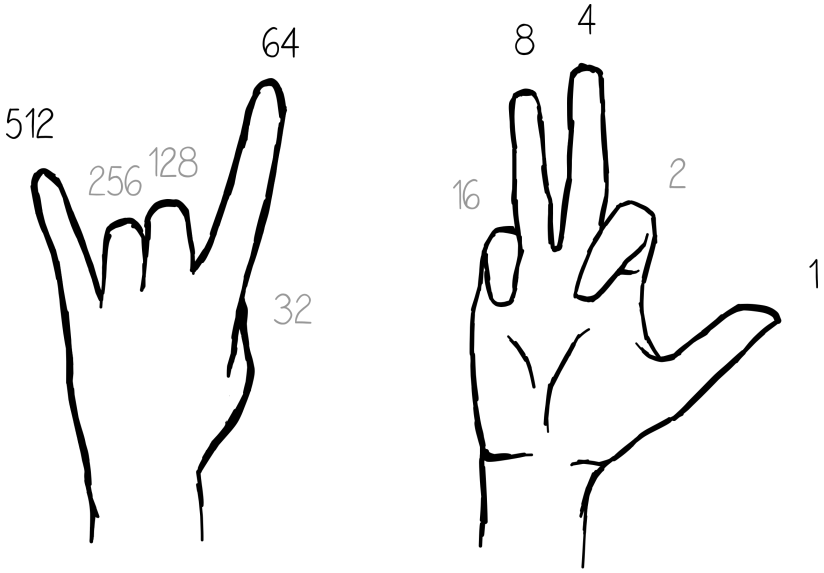




World of triangles

The 3D pictures seen in computer animated films or computer games consist of many small triangles. The shape of a figure in such an image comprises many points. To visualize a surface, a set of those points is connected by lines. The mathematical figure which emerges through this process is called polygon. The smallest polygon possible is the triangle, where only three points are connected. Furthermore, a triangle guarantees that all the points of the polygon are on the same plane.

The more triangles are used, the more detailed a surface is. Whereas a simple object with flat surfaces like a cube can be modeled by using only a few triangles, a more complex structure like a teapot requires many more triangles for representing its round shape.



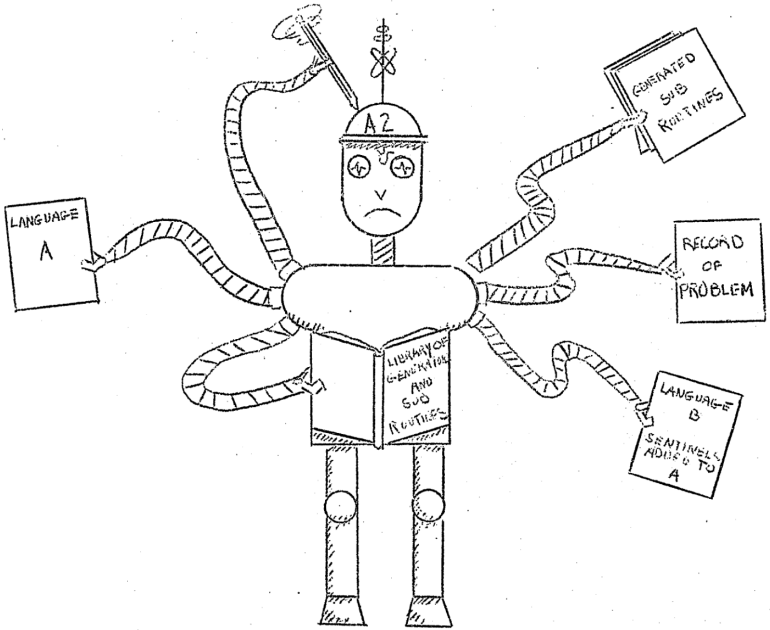
512 + 64 + 8 + 4 + 1 = 589

Counting to more than ten with two hands

Normally you count with your fingers by stretching them out one after another. You don’t care about which ones are stretched out, but only about the amount of them. Each of them is as significant as the other.
By changing the significance in the way that the right thumb has significance 1 and the left neighbor neighbor is always twice as significant we can get 1024 different numbers:

1+2+4+8+16+32+64+128+256+512=1023, and 0

This is called binary counting and exactly the way a computer counts but with zeros and ones instead of bent and stretched fingers.

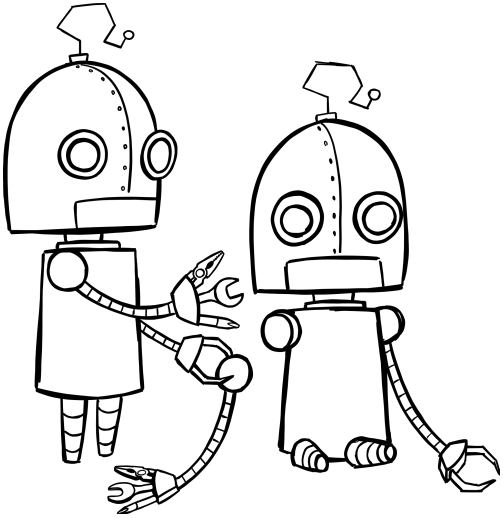

A2
LANGUAGE
A
GENERATED
SUB
ROUTINES
RECORD
OF
PROBLEM
LIBRARY OF
GENERATED
AND
SUB
ROUTINES
LANGUAGE
B
SENTINELS
ADDED TO
A

We have Machines within Machines

In 1954, computers just computed numbers. Grace Hopper, a famous female programmer around then, had a hard time to convince her colleagues that these machines can do more.

The picture is hers and shows how a computer program can translate one computer program into another one. She even coined the term that is still use today: a **compiler**.

Nowadays we have many more of these translation and transformation machines, with names as funny as the picture: *cross-compilers*, *transpilers* and *compiler-compilers* (cocos). Their mission is to translate from one computer language to another, for example to “compile” a human-written program to machine code, or to “transpile” from a new version of a programming language to the old one.



The fixpoint of describing yourself

What could be easier than to write a program that produces it’s own code as output? It’s not that easy – even a versed programmer will spend some hours to figure out the trick (which can be different for each programming language).

One trick is that the program must contain a description of itself, inside its own code. And because outputing that self-description also needs code, this self-description must be somehow compressed.

But once written, such a program (also called a “Quine”) reproduces itself each time it is run, and this in turn is called a fixpoint. One could call this the fixpoint of life, because any lifeform must master this trick, including robots if they wanted to build themselves and become a lifeform.

Seminar participants (spring semester 2019):

Fabricio Arend Torres
Nils Bühlmann
Rebecca Dold
Simon Dold
Omnia Kahla
Patrick Kahr
Marcel Lüthi
Sebastian Philipps
Linard Schwendener
Tim Steindl
Christian Tschudin
Marco Vogt
Moira Zuber

https://tilics.dmi.unibas.ch/