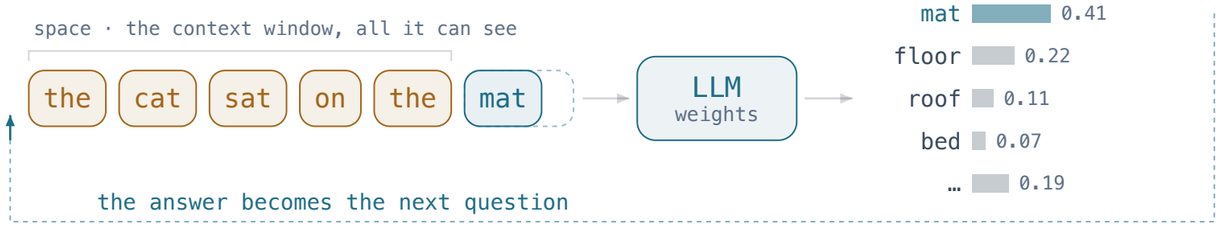
COMPILER CONSTRUCTION · WEEK 13 · STATION V

Generated Code

What a large language model is; a generated program as notation; the compiler, the test and the model checker as the gate.

WHAT IT IS

A large language model is a machine trained on notation, asked for meaning.



tokens in · a distribution over the next one out · then a sample

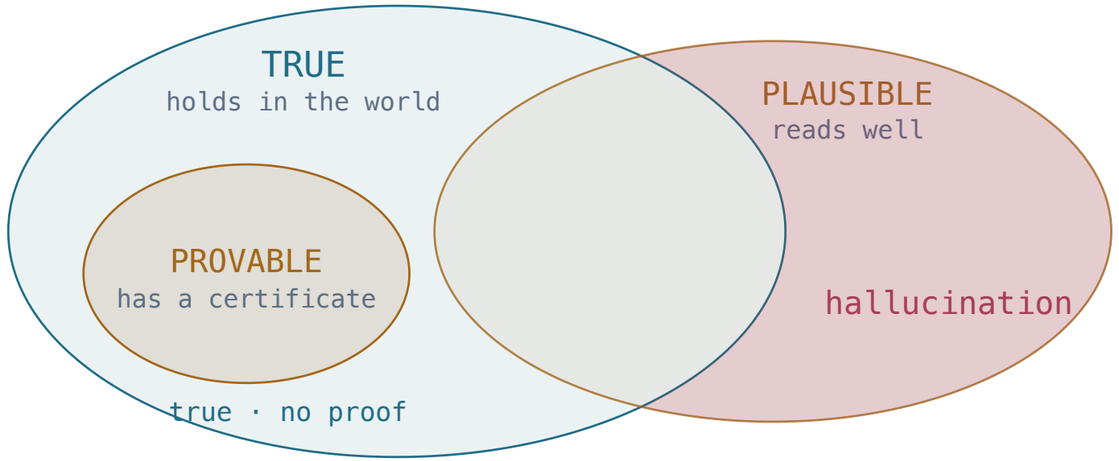
Context in, a probability for every next token out, one sampled and appended. Trained on text by predicting the next token: a purely syntactic signal. Meaning never handed to it.

It is arithmetic on bits — in principle a C* program, which mipster would run. It computes what a Turing machine computes, and week 8 said what that means about self-certification.

Its behaviour on every context is one of week 7's infinite sheets. Only sampled, never enumerated. Testing shows presence.

HALLUCINATION

A hallucination is a proof-shaped object that isn't true.



plausible · provable · true

Fluency is syntax. Correctness is semantics. A generated program can be perfectly fluent C — indented, idiomatic, commented — and not parse as C*, or parse and divide by zero, or run and compute the wrong thing. Three tests, and they are not the same test.

Not a defect to patch away: the gap of week 7 at consumer scale. The response is the same as everywhere in this class: check the notation against a semantics you own.

THE GATE · IN THIS CLASS

Ask a machine for a C* program. Then starc, mipster, rotor.

```
$ ./selfie -c generated.c
./selfie: syntax error in generated.c in line 3: unexpec
// … after asking again for C*, not C:
$ ./selfie -c generated.c -m 1 7
5040
$ ./rotor -c generated.c - 0 && tools/bitme.py -kmax 100
… core-0-division-by-zero — satisfiable · input 0
```

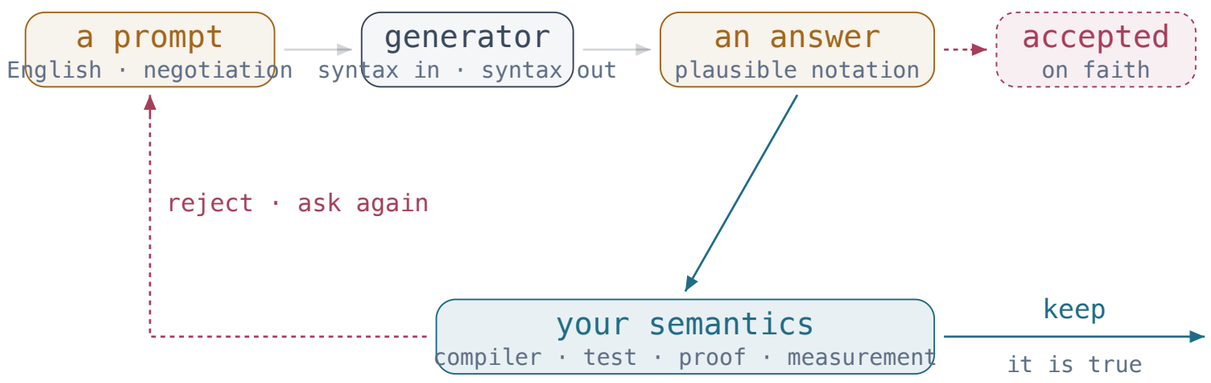
Does it parse? The compiler decides, always. Usually not on the first try: the machine produces C fluently, and C* is not C.

Does it do what was asked? Run it on the inputs you thought of. Presence, not absence.

Can it fail? Within a thousand steps, on any input: rotor and bitme. Absence, within the bound. None of the three is available to the machine that wrote the program, because the only semantics in the loop is the one you brought.

THE FOURTH APPEARANCE

A generator supplies notation. The semantics has to come from somewhere else.



what the machine supplies · and what it does not

Cantor's list needed an object built from outside it. Gödel's system needed a stronger one. Thompson's compiler needed a second compiler. The generator needs a check it cannot be. One theorem, four costumes, and this class met all four.

Models grading models, models trained on model output, agents invoking themselves: the loops are closed, and week 8 said what no loop establishes about itself. Spend the effort on the outside check.

WHAT CHANGED

Generation got cheap. Verification did not. This class was about verification.

Week 11: hard to find, easy to check. That is a description of a healthy relationship with a machine: accept work you could not have produced, provided you can check it. When you cannot, delegation is faith.

Value migrates to the two ends the machine does not occupy: stating what should be true, and establishing that it is. A grammar, a type, a test, a bad state and a bound. Every one of those is something you wrote this semester.

FOR THE CAPSTONE

Use a machine to draft struct-execution if you like. Then run the three checks, and note in your submission which of them found what. That note is worth more than the draft.