COMPILER CONSTRUCTION · WEEK 14 · STATION VI

What is a Compiler?

The sentence of week 1, earned. The definition of the subject. And where a compiler class leaves you.

THE SENTENCE, EARNED

A compiler is a proof system for syntax and a constructor of semantics — and every semantic question it seems to answer is a chosen approximation.

PROOF SYSTEM

Weeks 2–4. The scanner decides a regular language, the parser a context-free one, and a parse is a derivation, which is a proof. The symbol table and the types are attributes of the text; checking them is proof checking. All of it terminates, all of it answers.

CONSTRUCTOR

Weeks 5–8. Each rule's meaning built from its parts' meanings: literals, terms, statements, procedures, compositionally, into RISC-U, into a number. Assignment and while made the language universal — and the compiler blind to behaviour.

APPROXIMATION

Weeks 9–12. Every optimisation a syntactic stand-in for a semantic fact; every check sound or complete, never both; and the best anyone can do: the program as a formula, a solver, a bound. Applied to the compiler itself.

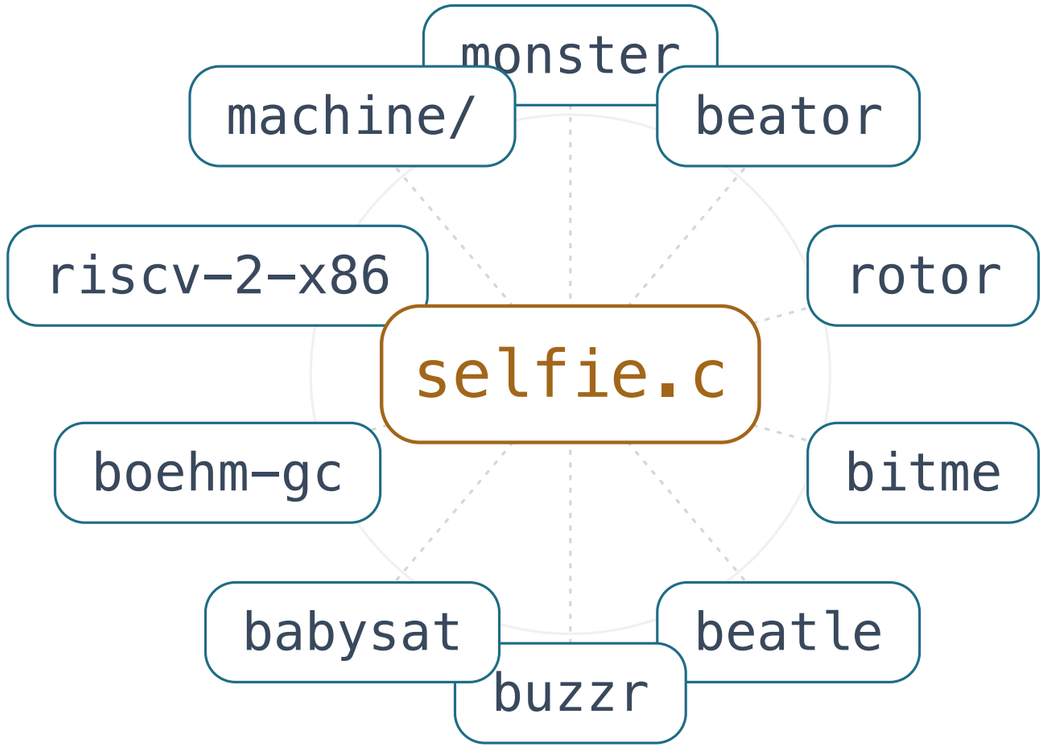
AND THE SUBJECT

So — what is computer science?

WORKING DEFINITION

> *Computer science is the exact study of notation a machine can execute — what can be written down, what can be computed, what can be decided, and what can be afforded — and therefore of the gap, measured precisely, between notation and meaning.*

Four verbs, and a compiler class touched every one: written down is the grammar, computed is the code generator, decided is the parser and the type checker, afforded is the solver. The gap is what the fixed point cannot close and the bound cannot cross.



the specimen, and the workshop around it

INTELLIGENCE

Discovering and understanding promising unproven truth — then building the notation that proves it.

Every extension you wrote this semester began with a grammar rule: a small new notation, decided before it was implemented, checked by a machine after. That is the definition of the introduction class at the scale of a homework, and it is what a language designer does at the scale of a career.

Machines are formidable at the second half — proving, generating, compiling. The first half, deciding what the notation should be and what it should mean, is not a step inside any system. It is the step to a new one, and nothing hands it to you.

WHERE THIS LEAVES YOU

You can read a language definition and know what it commits to. You can read a compiler and know which of its answers are proofs and which are guesses. You can hand a program to a solver and read the answer. Those three do not depreciate.

BEYOND THE CLASS

The workshop is open.

Rotor's synthesis models let a solver write RISC-V code. Its equivalence models let it check that starc's output and gcc's compute the same thing. Bitme's decision diagrams are a research project. The systems class takes hypster where this class took starc.

Several tools on the galaxy figure are Master's and PhD work that started as a semester assignment. If a week of this class made you strangely comfortable, that is the selection criterion.

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*
>
> DONALD E. KNUTH, 1977 — AND EVERY COMPILER ENGINEER SINCE