INTRODUCTION TO COMPUTER SCIENCE · WEEK 4 · STATION II

Notation

Formal languages, and the three that selfie is made of: EBNF, C*, and RISC-U.

SYNTAX AND SEMANTICS

The score is not the music.

SYNTAX · NOTATION

Marks on paper. Finite, discrete, checkable, copyable. 1 + 1, a staff of crotchets, H_2O, a line of code, this sentence.

SEMANTICS · MEANING

What the marks are *about*. The number two. Sound in a room. A molecule that dissolves salt. A machine's behaviour over all inputs.

A semantics is a function from notation to meaning. Defining that function precisely is the founding act of every exact discipline — and computer science is where it was done most sharply, because a machine does exactly what the marks say and nothing else.

Frege 1892 · Sinn / Bedeutung

Saussure · signifier / signified

Korzybski 1931 · the map is not the territory

Magritte 1929 · ceci n'est pas une pipe

TWO KINDS OF LANGUAGE

Natural language is compression. Formal language is commitment.

"I saw the man with the telescope." Two readings, one string. English resolves it with a shared world you and I already have. That shared world is why English is powerful — and why it cannot be a foundation. Its semantics is *us*, and we differ.

A formal language pays a price — narrowness, pedantry, effort — to buy one thing: a meaning that does not depend on who is reading.

NATURAL	FORMAL
ambiguous	single-valued
elastic, forgiving	brittle, exact
persuades	proves
needs a mind	needs a machine
learned by living	defined by decree

Prompting an AI in English is negotiation. Writing a test, a type, a schema, a unit, a contract is legislation.

THE NOTATION FOR NOTATION

EBNF says what code may look like. It can even say what EBNF looks like.

```
integer = digit { digit } .
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6"
while = "while" "(" expression ")"
( statement | "{" { statement } "}" ) .
```

= defines. | is *or*. { } is *any number of*, including none. [ ] is *optional*. ( ) groups. Quotes hold the literal characters. A full stop ends the rule.

So an integer is a digit followed by any number of digits: 0, 42, 007. A while statement is the word, a parenthesised expression, and either one statement or a braced block of them.

The synopsis of week 2 was this notation's cousin, a regular expression: braces, brackets, bars, and nothing else.

C*

7 keywords. 22 symbols. One data type, and pointers to it.

`uint64_t` `void` `sizeof` `if` `else` `while` `return`

Five statements: assignment, while, if-else, procedure call, return. Arithmetic `+ - * / %`, comparison `== != < > <= >=`, the `*` that reads a pointer — hence the star in the name. No other types, no arrays, no Boolean operators.

It is a real subset of C: any C compiler compiles C*. And it is enough to write a compiler for itself, an emulator, and a hypervisor, which is the only argument for its size.

```
// tiny.c
uint64_t c;
uint64_t main() {
c = 0;
while (c < 7)
c = c + 1;
return c;
}
```

RISC-U

14 instructions. 32 registers. 4 GB of memory. A real subset of real RISC-V.

GROUP	INSTRUCTION	WHAT IT DOES
initialise	`lui rd,imm` · `addi rd,rs1,imm`	put a number into a register
memory	`ld rd,imm(rs1)` · `sd rs2,imm(rs1)`	load a word from memory, store one
arithmetic	`add` · `sub` · `mul` · `divu` · `remu`	rd = rs1 ∘ rs2, unsigned
comparison	`sltu rd,rs1,rs2`	rd = 1 if rs1 < rs2, else 0
control	`beq rs1,rs2,imm` · `jal rd,imm` · `jalr rd,imm(rs1)`	branch if equal, jump and link, jump to a register
system	`ecall`	ask the operating system for something

Every instruction is 32 bits. Every one of them also does `pc = pc + 4`, or says where `pc` goes instead. That is the entire machine language, and the machine chapter gives each line its exact meaning.

THREE NOTATIONS, ONE MEANING

What the compiler makes of the while loop.

```
$ ./selfie -c tiny.c -S tiny.s
./selfie: 432 bytes generated with 104 instructions and 16 bytes of data
$ cat tiny.s
…
0x158(~6): 0xFF01B283: ld t0,-16(gp) // c
0x15C(~6): 0x00700313: addi t1,zero,7 // 7
0x160(~6): 0x0062B2B3: sltu t0,t0,t1 // c < 7 ?
0x164(~6): 0x00028C63: beq t0,zero,6[0x17C] // if not, leave the loop
0x168(~7): 0xFF01B283: ld t0,-16(gp) // c
0x16C(~7): 0x00100313: addi t1,zero,1 // 1
0x170(~7): 0x006282B3: add t0,t0,t1 // c + 1
0x174(~7): 0xFE51B823: sd t0,-16(gp) // c = …
0x178(~9): 0xFE1FF06F: jal zero,-8[0x158] // back to the condition
```

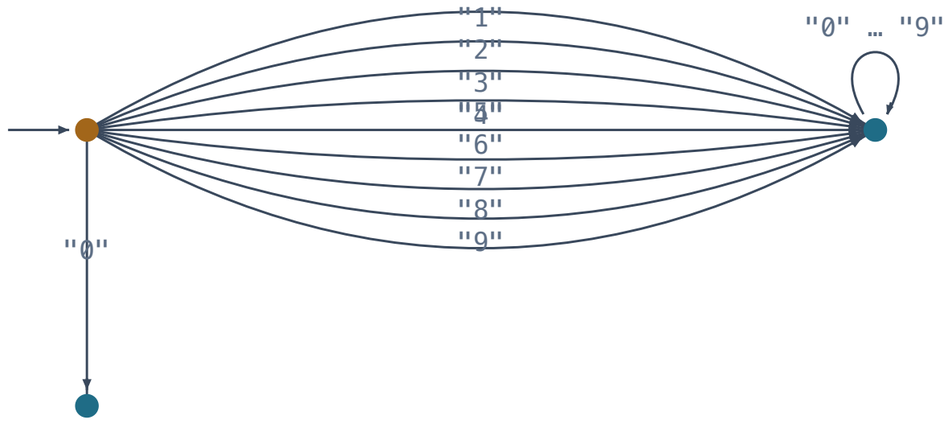
Source. `while (c < 7) c = c + 1;` — for you.

Assembly. Nine instructions with names — for the person reading the machine.

Bits. `0xFF01B283` and eight more words — the only thing the machine sees. Three notations. One meaning. The last one is the one that runs.

WHAT A MACHINE CAN CHECK

A finite machine decides a regular language. A stack decides a context-free one.




```
integer        = "0" | non_zero_digit { digit } .
non_zero_digit = "1" | … | "9" .
digit          = "0" | non_zero_digit .
```

integer literals without leading zeros

A finite state machine reads one character at a time and remembers only which circle it is in. That is enough to check integers, identifiers, and the whole synopsis — anything a regular expression describes.

It is not enough for parentheses: to check that every (has its) you must count without bound, and a finite machine cannot. Add a stack and you can: expressions, statements, procedures, all of C*, all of EBNF itself.

Regular inside context-free inside … — the sizes of notation form a ladder, and week 9 says where the ladder ends.

STATION II

Everything on this deck is a finite sequence of symbols. Hold that thought.

A grammar is a finite text. A program is a finite text. The binary the compiler produced is a finite sequence of bytes. The machine language is a table with fourteen rows.

Finite things from a finite alphabet can be *listed*: all of length one, then two, then three. Next week we do the listing, and it turns out that every program that will ever be written has a number — and that a compiler and an emulator are just arithmetic on those numbers.

THE GAP, SO FAR

Week 2: one prefix, two meanings. Week 3: one byte, five. Week 4: three notations, one meaning, fixed by decree. The decree is a program, and it is written in one of the three notations. That loop is the subject of week 8.

BEFORE NEXT WEEK

Recommended exercises.

1. Read the Notation chapter: C*, RISC-U, EBNF.
2. Write the EBNF rule for a C* character literal, and for a string literal. Check against `grammar.md`.
3. Derive `c = c + 1;` from the grammar, rule by rule, starting at `statement`.
4. Type `tiny.c`, compile it with `-S`, and find the nine instructions of the loop. Change 7 to 700 and see which bits change.
5. Write a regular expression for the dates of this course, and an EBNF grammar for nested parentheses. Which of the two needs the stack?

NEXT WEEK

Countability: everything you can write down can be listed, so every program has a number — and a binary is one number, 188,392 bytes long. The compiler compiles itself and gets the same number twice.