COMPILER CONSTRUCTION · WEEK 3 · STATION II

The Parser

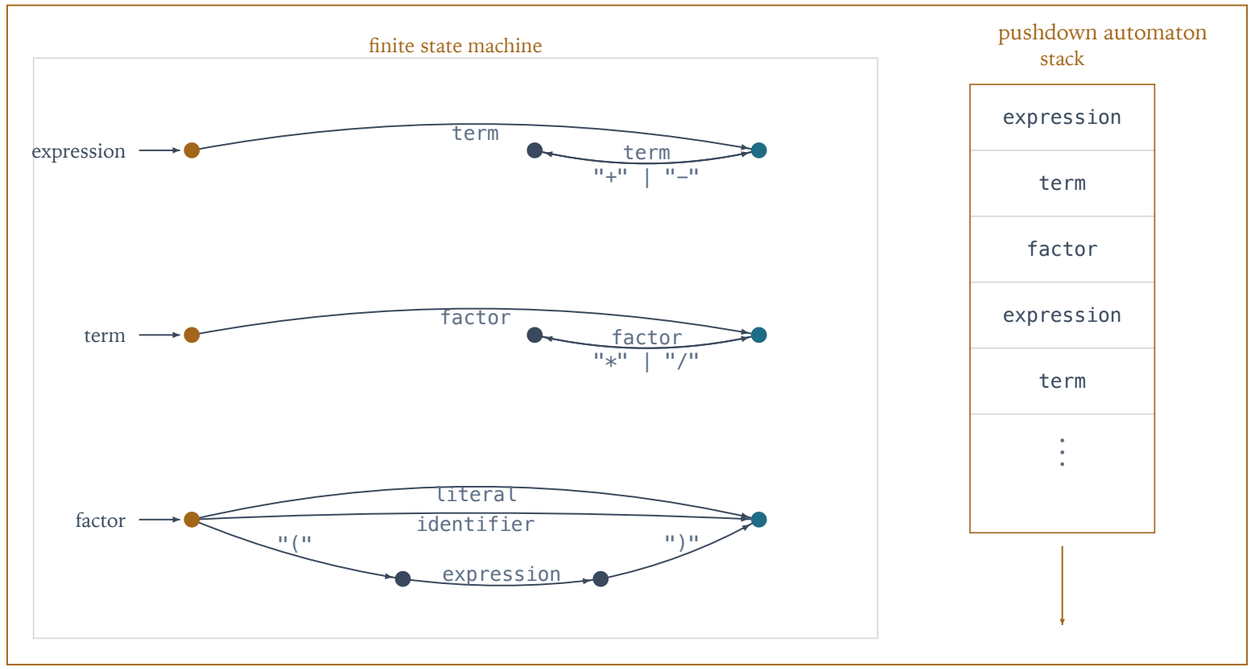
Context-free grammars, LL(1), recursive descent — and why a parse is a proof.

BEYOND REGULAR

A finite machine cannot count. Parentheses need counting.

```
expression = term { ( "+" | "−" ) term } .
term       = factor { ( "*" | "/" ) factor } .
factor     = literal | identifier | "(" expression ")" .
```

context-free grammar



a context-free grammar, one FSM per rule, and a stack

Whatever number of states a machine has, feed it more opening parentheses than that and two different depths land in the same state. So no finite state machine checks that every (has its). That is the pumping lemma in one sentence.

The C* grammar has rules that mention each other — `factor` mentions `expression` which mentions `term` which mentions `factor` — and cannot be substituted into one rule. It is context-free, not regular.

The machine that decides it is a finite state machine plus a stack: a pushdown automaton. One FSM per rule, and the stack remembers which rule to return to.

LL(1)

Left to right, leftmost derivation, lookahead of one.

C* is LL(1): read the symbols from left to right, always expand the leftmost non-terminal, and decide which alternative of a rule applies by looking at one symbol. The grammar was designed so that this works — the same principle as the scanner's lookahead, one level up.

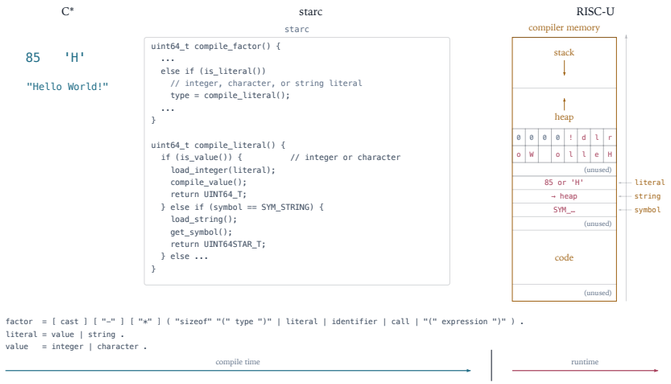
LR(1) languages need to be parsed from the right-hand sides backwards and are harder; most production compilers use generated parsers for them. LL(1) can be parsed by hand, with joy, and industry engineers do it routinely: simplify the syntax until it is LL(1), then write the parser.

RECURSIVE DESCENT

One procedure per rule, named `compile_X` for non-terminal `X`. A non-terminal on the right-hand side is a call. A terminal is a check against `symbol` and a `get_symbol()`. `{ }` is a while, `[ ]` is an if, `|` is an if-else chain. The grammar *is* the program.

IN SELFIE

Parsing literals: `compile_factor`, `compile_literal`, `compile_value`.



compile time · and what the runtime will see

A literal is used in exactly one place in the grammar, inside `factor`. So `compile_factor` looks at one symbol: if it is a value or a string, it calls `compile_literal`, which consumes it and returns its type — the first grammar attribute of the class.

Left of the bar: compile time, the compiler's memory. Right: runtime, the target's. Compile time is also runtime — for the compiler's own machine code — and keeping the two apart is the single most common difficulty in this class. Read that twice.

PRECEDENCE AND ASSOCIATIVITY

The grammar encodes precedence. The code encodes associativity.

```
expression = term { ( "+" | "-" ) term } .
term = factor { ( "*" | "/" | "%" ) factor } .
factor = literal | identifier | "(" expression ")" .
```

`x + 7 * y` groups as `x + (7 * y)` because `*` lives one rule deeper than `+`: lower precedence, higher in the grammar. That is syntactic, and EBNF can say it.

`x - 7 + y` groups as `(x - 7) + y` because the while loop in `compile_expression` emits code as it goes, left to right. That is associativity, and EBNF cannot say it — it shows up only in the implementation.

Even `+` is made left-associative, although addition associates in arithmetic. Why? Because it does not associate on a machine with overflow, and week 3 of the introduction class said why.

SYNTAX ERRORS

Report, continue, and finish the program anyway.

A compiler that stops at the first error is a compiler you run a hundred times. So every `compile_X` begins by synchronising: skip symbols until one arrives that can start an `X`, report what was skipped, and go on. The generated code may then be nonsense, and that is fine; nobody runs it.

Designing good error handling is more art than science, and selfie's is minimal: a message with the line number and the unexpected symbol. It is also, as week 8 will note, the compiler deciding something — about the text.

```
$ printf 'uint64_t main() { return 1 + ; }' > bad.c; ./
./selfie: syntax error in bad.c in line 1: unexpected s
./selfie: syntax error in bad.c in line 1: unexpected s
```

STATION II

A parse is a proof.

A derivation — start symbol, rule, rule, rule, down to the terminals — is a finite object that anyone can check step by step. That is what a proof is. A successful parse constructs one, and the parse tree is the proof written down.

And the parser always terminates: every call consumes symbols or returns, and there are finitely many. Membership in a context-free language is *decidable*. Everything the parser says is about the text, and everything about the text can be decided.

WHAT THE PARSER DOES NOT KNOW

Whether x was declared. Whether the types match. Whether the program halts. The first two are next week, and they are still about the text. The third never is.

SIZES OF NOTATION

Regular: one rule, no memory. Context-free: many rules, a stack. Above that, grammars a machine can only semi-decide, and above that, the halting problem. Chomsky's ladder is the axis of this class, on the countable side.

THIS WEEK

Finish `hex-literal`. Read ahead.

1. Finish and submit `hex-literal`: grammar first, then the scanner, then `self-compile`.
2. Translate the `while` and `if` rules into recursive-descent procedures on paper, before looking at `compile_while` and `compile_if`.
3. Derive `x = x + 7 * y;` from the grammar starting at `statement`, and draw the tree. Mark which node fixes the precedence.
4. Read the book's sections on Variables and the Symbol Table.

NEXT WEEK

Symbols and types: the symbol table, what a type is, and why type checking is proof checking while almost everything else about meaning is not. Assignment: bitwise shift operators, compilation.