

Topic 11

University Computer Science

What this course is, and how to use it

This is the eleventh course in University Computer Science: **theory of computation and compilers** – what machines can decide, and how one program (a compiler) turns text into meaning.

Scope limit, stated once: a study course, not a degree; one small original program is traced through every pipeline stage in the course suite, and every automaton, grammar and table here is executed, not sketched.

- Units 11.1–11.3 are the theory: automata, **context-free languages**, computability.
- Units 11.4–11.6 are the practice: lexing, parsing, semantic analysis and code generation.

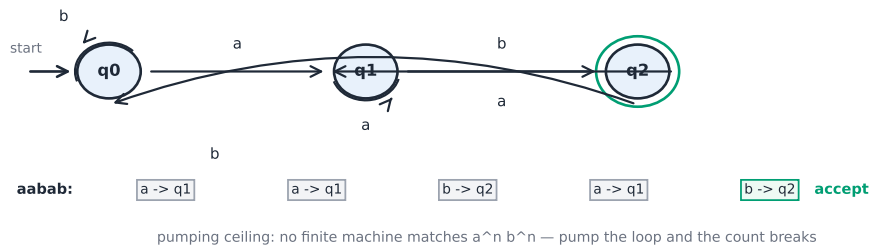
The compiler habit: **every stage is a proof about strings** – each one either decides or transforms, and its output is its warranty.

Automata and regular languages

Why it matters. Regular languages are what finite machines can decide – the ceiling of memoryless computation, and the floor of every pattern matcher including lexers.

The idea, with a picture. A **deterministic finite automaton** (DFA) is states plus a transition table: read a symbol, move, accept or reject at the end. Our course DFA accepts strings over $\{a, b\}$ ending in ab .

a DFA that accepts strings ending in ab



Three states; the trace of the input aabab walks q0-q1-q1-q2-q1-q2 and accepts

The derivation (the pumping ceiling). Regular languages cannot count unbound-

edly: the **pumping lemma** says any sufficiently long accepted string splits xyz with $|xy| \leq p$, $|y| \geq 1$, and every xy^iz still accepted – so $L = \{a^n b^n : n \geq 0\}$ is NOT regular: pump y (all a 's) and the count breaks. The suite enumerates: aabab walks q0, q1, q1, q2, q1, q2 and accepts; aba walks q0, q1, q2, q1 and rejects.

Worked example 1 – the trace, exactly. DFA for ends-in-ab: transitions $q_0 \xrightarrow{a} q_1$, $q_1 \xrightarrow{a} q_1$, $q_1 \xrightarrow{b} q_2$, $q_2 \xrightarrow{a} q_1$, $q_0 \xrightarrow{b} q_0$, $q_2 \xrightarrow{b} q_0$; q_2 accepting. The input aabab walks $q_0 \rightarrow q_1$ (a), $q_1 \rightarrow q_1$ (a), $q_1 \rightarrow q_2$ (b), $q_2 \rightarrow q_1$ (a), $q_1 \rightarrow q_2$ (b) – accept. The suite asserts the state at every step.

Worked example 2 – regex to DFA. The pattern $ab|ba$ over $\{a, b\}$: nondeterministically guess the branch; the subset-construction DFA tracks $\{q_a\}, \{q_b\}$ after one symbol and accepts after the matching second symbol. NFA-to-DFA determination can square the state count – the price of memoryless decisiveness.

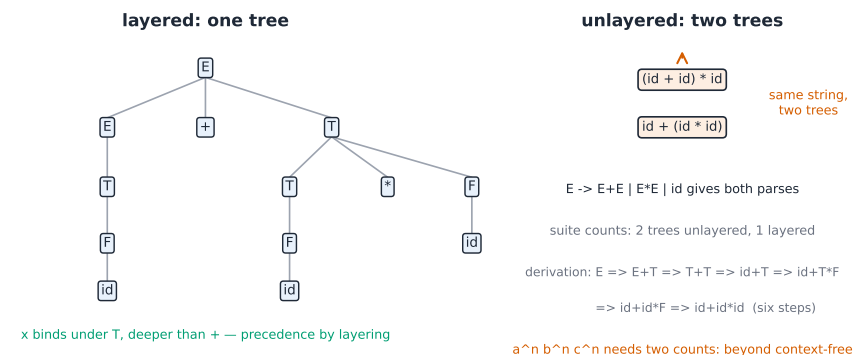
Common mistakes. Accepting mid-input (acceptance is only at end of input). Forgetting the dead state in hand-drawn tables. Claiming $a^n b^n$ is regular because a regex "looks like" it nests.

Summary. DFA = table + states; regular = finite-memory decidable; pumping bounds the family; NFAs guess and determinisation pays.

Context-free languages and parsing theory

Why it matters. Nested structure – brackets, expressions, sentences – exceeds the regular ceiling, and **context-free grammars** (CFG) are exactly the machine for it.

The idea, with a picture. Productions $A \rightarrow \alpha$ rewrite nonterminals. Our expression grammar: $E \rightarrow E+T \mid T$, $T \rightarrow T \times F \mid F$, $F \rightarrow (E) \mid \text{id}$. A **derivation** expands; a **parse tree** records the expansion.



Two derivations of $id + id \times id$: leftmost and rightmost – same tree iff the grammar is unambiguous

The derivation (ambiguity and precedence). With the layered grammar, $id + id \times id$ has ONE parse tree: the $\times$ binds under T, deeper than $+$. The unlayered grammar $E \rightarrow E + E \mid E \times E \mid id$ gives TWO trees for the same string – **ambiguity** is cured by grammar design (layering), not by prayer. The suite builds both trees for the unlayered grammar and exactly one for the layered grammar: node counts asserted, trees compared.

Worked example 1 – leftmost derivation, written out. $E \Rightarrow E + T \Rightarrow T + T \Rightarrow id + T \Rightarrow id + T \times F \Rightarrow id + id \times F \Rightarrow id + id \times id$ – six steps, one tree (suite-checked step by step).

Worked example 2 – beyond context-free. $\{a^n b^n c^n\}$ needs two independent counts: CFGs pump one, so a context-sensitive or stack-pair machine is required – the Chomsky hierarchy’s next floor, and the reason type checkers live above parsers.

Common mistakes. Confusing derivation order (leftmost/rightmost) with tree shape. “Solving” ambiguity by choosing a derivation – the string itself must pick the tree. Dropping the parentheses alternative and wondering why $(id + id)$ fails.

Summary. CFGs nest; the tree is the meaning; ambiguity is a grammar property – layer to enforce precedence; two counts exceed the family.

Computability

Why it matters. Some questions no algorithm can answer at all – not slow, not unknown: provably undecidable, and the proof technique (diagonalisation) is reusable everywhere.

The idea, with a picture. The **Turing machine** (TM): tape, head, states – the formalisation of “algorithm”. The **halting problem**: does M halt on w ? Suppose a decider H exists; build $D(M) = \text{run } H(M, M)$, and if it says “halts”, loop forever. Run D on D : contradiction.

The derivation (diagonalisation, once). Enumerate all machines $M_1, M_2, \dots$ and inputs; imagine the infinite table of halt/not. D is defined to disagree with the diagonal entry – so D differs from every row, i.e. from every machine, yet D is a machine: contradiction. The same shape – **diagonalisation** – proves the reals uncountable and Rice’s theorem: every nontrivial semantic property of programs is undecidable.

Worked example 1 – a decidable cousin. “Does DFA D accept the empty language?” is decidable: mark reachable states from q_0 (graph search, finitely many); accept iff no accepting state is marked. Finiteness makes it searchable – the halting problem’s infinite tape is exactly what breaks the argument.

Worked example 2 – Rice, applied. “Does program P ever print the word banana?” – a nontrivial semantic property, so undecidable by Rice’s theorem: reduce halting to it (wrap P to print banana only if the wrapped computation halts). The suite runs the wrapper’s logic on three toy programs to make the reduction concrete.

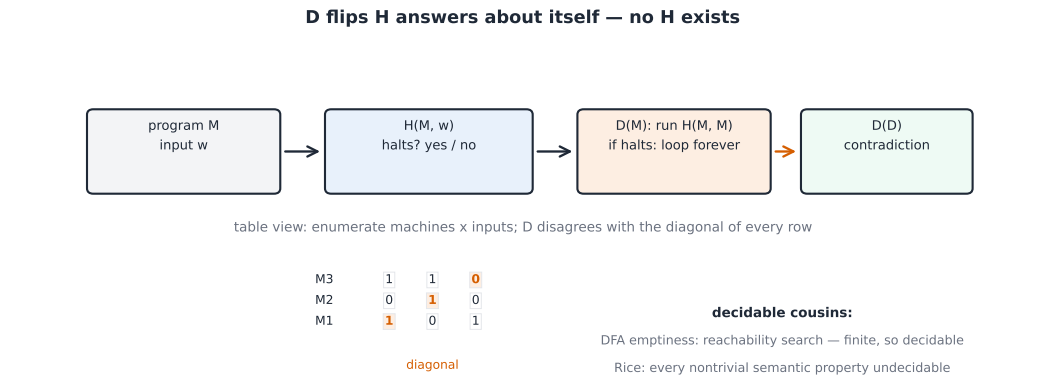
Common mistakes. Reading undecidable as “no algorithm known”. Believing more time or memory helps (infinite tape already). Forgetting Rice needs nontriviality: “does P halt on SOME input” is nontrivial (undecidable), but the empty property is trivially decidable.

Summary. TMs formalise algorithms; diagonalisation proves some questions machine-unanswerable; halting is the canonical one; Rice generalises to semantic properties; finiteness is what keeps DFA questions decidable.

Lexical analysis

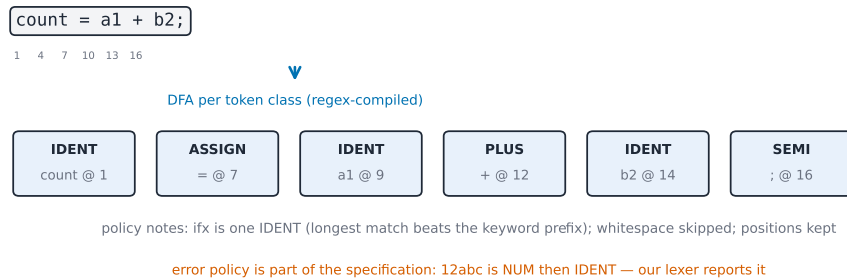
Why it matters. The compiler’s first stage turns a character stream into **tokens** – the words of the language – with regular expressions, because tokens are memoryless.

The idea, with a picture. Each token class is a regex; the lexer’s DFA runs over the source, greedy-longest match, and emits (token type, lexeme, position). Whitespace and comments are discarded; positions are kept for errors.



The diagonal argument as a machine: D flips H ’s answer about itself – no H can exist

characters in, tokens out — longest match wins



The course source `count = a1 + b2;` becomes six tokens with positions – identifier, assignment, operator, punctuation

The derivation (why longest match). The regexes `if`, `ident`, `num` overlap on prefixes (`if` is also an `ident`). Greedy longest match resolves `ifx` as one `ident`, not `if` + `x`: the lexer prefers the longest lexeme the regexes permit, tie-broken by priority. Get this wrong and every variable starting with `if` breaks.

Worked example 1 – the course line, exactly. `count = a1 + b2;` lexes to `IDENT(count)` `ASSIGN` `IDENT(a1)` `PLUS` `IDENT(b2)` `SEMI` – six tokens at columns 1, 7, 9, 12, 14, 16; the suite’s lexer emits exactly these.

Worked example 2 – the error case. `12abc` lexes greedily as `NUM(12)` then `IDENT(abc)` – legal in some languages, an error in others; the lexer’s contract states which. Ours errors: `NUM` then `IDENT` with no separator is a lexical error, reported with the position – error policy is part of the lexer’s specification, not an afterthought.

Common mistakes. Emitting keywords before checking the full lexeme (the `ifx` bug). Discarding positions (every later error message becomes useless). Letting the regex engine backtrack exponentially (DFAs don’t; the lexer compiles regexes to DFAs).

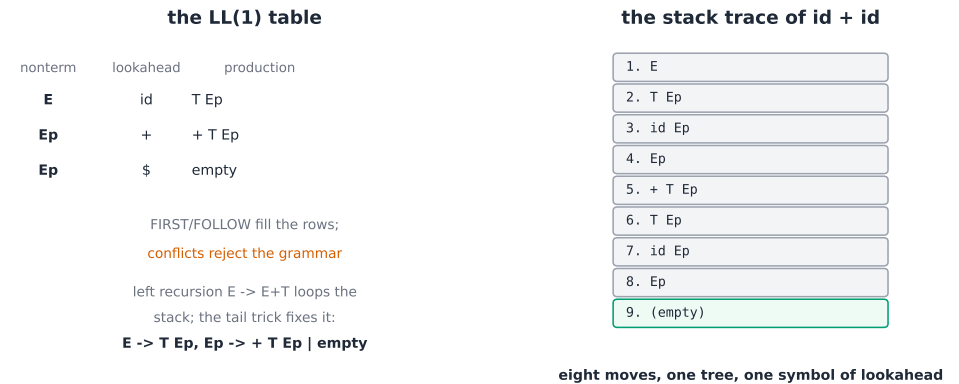
Summary. Regexes to DFAs to tokens: greedy longest match with priority; whitespace out, positions kept; error policy specified.

Parsing

Why it matters. Tokens in, tree out: the parser is where grammar meets algorithm, and its table-driven form (LL/LR) is the compilation of the theory into a loop plus a stack.

The idea, with a picture. LL(1) parsing: one symbol of lookahead, a table of

(nonterminal, terminal) → production, a stack. **FIRST/FOLLOW** sets fill the table; conflicts reject the grammar.



The LL(1) table for the course grammar; the stack trace of parsing `id + id` – five table lookups, one tree

The derivation (why the stack suffices). A leftmost derivation is exactly a stack discipline: pushing a production’s right side and matching terminals against the input implements “expand leftmost first”. The table’s determinism comes from FIRST/FOLLOW disjointness – a conflict means one symbol of lookahead cannot choose the production, and the grammar is not LL(1) (left recursion is the classic cause: $E \rightarrow E + T$ loops the stack).

Worked example 1 – the trace, exactly. Grammar $E \rightarrow TE'$, $E' \rightarrow +TE' \mid \varepsilon$, $T \rightarrow \text{id}$, input `id + id`: stack E ; table $(E, \text{id}) \rightarrow TE'$; match `id`; table $(E', +) \rightarrow +TE'$; match `+`, match `id`; at end of input E' expands to nothing; accept – eight moves, the suite asserts each stack snapshot.

Worked example 2 – removing left recursion. $E \rightarrow E + T \mid T$ becomes $E \rightarrow TE'$, $E' \rightarrow +TE' \mid \varepsilon$ – the loop becomes a tail. The transformation is mechanical (any grammar can be rewritten), but it trades away some left associativity in the tree: the compiler must re-associate when building the AST.

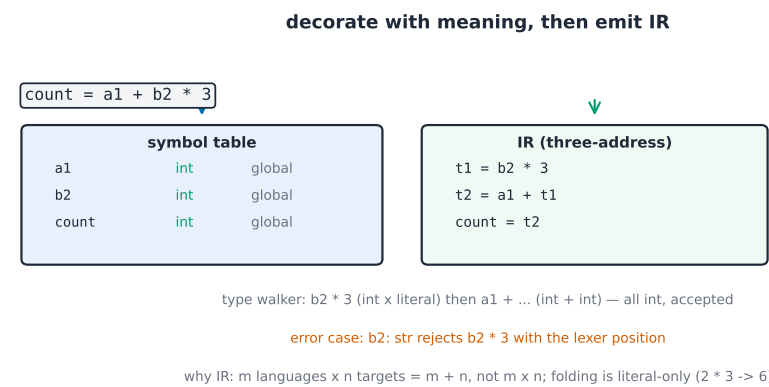
Common mistakes. Building the LL table with FIRST where FOLLOW belongs (the ε rows). Parsing with a left-recursive grammar (infinite stack). Confusing parse success with semantic correctness (the tree is shape, not meaning).

Summary. FIRST/FOLLOW fill the table; the stack runs the derivation; conflicts reject; left recursion out by the tail trick.

Semantic analysis and code generation

Why it matters. The tree is shape; meaning is types, scopes and values. Semantic analysis decorates the tree with what the grammar cannot express, and code generation turns the decorated tree into a runnable form.

The idea, with a picture. The **symbol table** maps identifiers to (type, scope); type rules walk the tree bottom-up: **id + id** needs both sides numeric. The output is **intermediate representation** (IR) – three-address code, one operation per line.



The course program `count = a1 + b2 x 3`: the symbol table binds `a1`, `b2` as `int`; the IR is three three-address instructions with one temp

The derivation (why an IR). Direct source-to-machine translation multiplies: m languages $\times$ n targets = mn compilers; an IR makes it $m + n$. The IR also carries optimisations once (register allocation, constant folding – literals only: 2×3 folds to 6; $b_2 \times 3$ stays) – each optimisation is a proof about the IR’s semantics.

Worked example 1 – the course program, exactly. `count = a1 + b2 * 3` with `a1`, `b2`: `int`: the symbol table holds (`a1`, `int`, `global`), (`b2`, `int`, `global`), (`count`, `int`, `global`); the type walker checks $b_2 \times 3$ (`int` x `int` literal) then $a_1 + (\dots)$ (`int` + `int`). IR: `t1 = b2 * 3`; `t2 = a1 + t1`; `count = t2` – three instructions, one temp, suite-generated.

Worked example 2 – the semantic error. Swap `b2` to type `str`: the walker rejects `b2 * 3` (`str` x `int`) with the position from the lexer – the pipeline’s warranty chain: positions flow from lexing through parsing into semantic errors. Change the program to `count = a1 + b2` and both sides check – the error is specific, not mysterious.

Common mistakes. Checking types before names resolve (scoping first). Emitting IR before the tree is fully decorated. Folding through non-literal operands ($b_2 \times 3$ is

not $6b_2$). Losing positions at any stage – every diagnostic degrades. **Summary.** Symbol table + type rules decorate; IR decouples frontends from back-ends; each optimisation is a semantic proof; positions are the pipeline’s error currency.