Lexing Define tokens with regular expressions (finite automata)
Disambiguate with longest match, rule priority, white space

CFG A quadruple (N, T, R, S):
N – non terminals
T – terminals
R – rules, $R \subseteq N \times (N \cup T)^*$
S – start symbol, $S \in N$
Disambiguate with rewrite:
E ::= E + E | E * E | N ⇒
E ::= E + F | F, F ::= F * N | N

LL(k) Recursive descent parser
For each non terminal compute set of terminal symbols that can begin strings derived from X, set of symbols that can follow X
Eliminate left recursion with factoring: E ::= T | E + T ⇒
E ::= T E′, E′ ::= + T E′ |
Could be conflicts in the predictive parse table (showing possible derivation rules for current non-terminals), so not in LL(k)

LR(k) Postpone production selection until entire right hand side of production rule has been seen

LR(0) Parser

Consists of a stack (list of states, topmost state is the current one), action table (to which new state it should move), goto table (a grammar rule to apply given the current state and current symbol in the input stream)

1. Initialize stack with [0]
2. Lookup action by state, input terminal:
    a. Shift: advance input stream and push char, new state onto stack
    b. Reduce: push reduced rule, for each symbol on RHS of rule remove a state from the stack, lookup in goto table by current state and LHS of rule and push it onto the stack
    c. Accept: terminate
    d. No action: error out

LR(0) Parser Generators

LR(0) items: if the current state contains the item A ::= α•cβ and the current symbol is c then shift (next state is A ::= αc•β), if the current state contains the item A ::= α• then reduce, A ::= α•Xβ is the tricky case (ε-trans)
Can use these to represent parser as a NFA:

1. Each LR(0) item is a state
2. Transition from A ::= α•cβ to A ::= αc•β with label c (c [non-]terminal
3. Transition from A ::= α•Xβ to X ::= •γ with label ε (X non terminal)
4. A ::= α• is **a** final state (i.e. reduce)
5. Obvious start state

Build a DFA from the NFA by:

1. Create rule S ::= A$
2. Create first state Closure({S::= •A$})
3. Pick a state I, for each item A ::= α•Xβ in I find Goto(I, X), add it if it is not already a state, and add an edge from I to Goto(I, X)
4. Repeat step 3 until no more additions

Note: Goto(I, X) is the set of LR(0) items in I that can be got by moving the • over X
Construct the goto and action tables: shifts are terminal-labelled edges, gotos are non-terminal labelled edges and reductions are accepting states in the DFA (those containing an item of the form A ::= α•): note conflicts!

LR(1) Parsers Reaction to problems with LR(0): unnecessary conflicts

LR(1) items: A pair of a LR(0) item & terminal (lookahead terminal, follows the production)
Modify closure operation so that it closes in a production for every possible first symbol
Now a state in the DFA that contains [X ::= α•, b] is recorded in the table as "reduce on lookahead b": allows disambiguation in parse!

LALR Relies on the observation that often a reducing state contents can be grouped by the derivation rule part, with a number of lookaheads for each one
10 times fewer states

Parse Tree A derivation tree based on the actual grammar rules

AST Contains only the information needed to generate an intermediate representation

Scope Scope: range of statements over which a variable visible
L-value: memory location
R-value: value stored at location
Static and dynamic binding

Stack Machine Pop, popto, push, pushfrom, swap, arith, goto, test, load, store

Register Machine Registers, memory locations, immediates
Can do simple direct stack translation to emit register code

Nested Functions How do you access stack allocated variables in functions you are nested in?
Dijkstra Displays: each stack frame contains pointers to all necessary frames at a lower nesting depth (uses space, slows function call, but runtime better)
Single Static Link: Each frame contains a single static link to the most recent frame at a lower nesting depth (less space, but runtime must chase pointers)
Lambda Lifting: Explicitly expand functions to take all free variables as arguments (but lots of duplication of values on the stack)
Closures: heap-allocated list of function pointer and free variables (necessary for using functions as values)

Optimisation Inlining small functions
Constant folding
Unused variable elimination
Direct function calls as a special case of closures

Object Files Symbols exported, imported
Relocation information

Memory Explicit memory management (potentially better but hard)
Garbage collection: use root set (stacks, registers) to identify reachable objects and reclaim unused ones
Reference counting: can be costly (memory access), can't detect cycles, incremental
Mark and sweep: do depth first traversal of object graph and add unmarked data onto free list (must only do this when there is enough garbage or GC cost is high), may use lots of stack (recursion), heap fragmentation
Copy collection: use two heaps, copy reachable data between them, simple, no fragmentation, but uses lots of memory and long GC pause
Generational: use copy collection for young generation, mark and sweep for the older ones, track pointers between generations, collect old generations infrequently

Objects Static calls: resolve function pointer by static type
Dynamic calls: resolve function pointer by object vtable
Subtyping: implement by class prefixing (but if using MI must add explicit pointer conversion)
Enforce visibility rules with errors during compilation

JVM Typed instructions
invokevirtual, invokeinterface, invokestatic, invokenonvirtual
Class loader: verification of type, allocation of class memory w/ default values, symbol resolution, initialization of class ctors
Bytecode interpreter, JIT compiler (method granularity), adaptive compiler (do advanced optimization for hottest methods)