Logic

1 PROPOSITIONS & CONNECTIVES

proposition - a statement that is either true or false ("the list is empty", "x > 5" for a known x); not questions, commands, opinions. In code, any boolean expression

the connectives - and (both), or (at least one, inclusive), not (flip), implies, iff (both the same). JS: &&, ||, !; iff for booleans is a === b, xor (exactly one) is a !== b

implies (A -> B) - false in exactly one case: A true and B false. A promise about when A holds; A false = promise never tested = true. Not causation

implication as code - A -> B rewrites to !A || B. "every admin must be verified" is !isAdmin || isVerified; the enforcing guard is the rule's negation: if (isAdmin && !isVerified) throw

truth table - one row per input combination, 2^n rows for n booleans. Two conditions are interchangeable exactly when their tables match on every row - the refactoring safety net

De Morgan's laws - not(A and B) = (not A) or (not B); not(A or B) = (not A) and (not B). Negating !(isValid && isReady) is !isValid || !isReady, not !isValid && !isReady

if-then's three relatives - "if A then B" (A->B): converse B->A (NOT equivalent), inverse notA->notB (NOT equivalent), contrapositive notB->notA (ALWAYS equivalent). Only the contrapositive matches the original's truth

vacuous truth - "if A then B" is true whenever A is false, regardless of B. [].every(x=>x>1000)===true; "all users in this empty segment are premium" is technically true, practically meaningless

necessary vs sufficient - necessary: required but not enough alone (a ticket to board a flight). Sufficient: enough alone but not the only way (born in the US, for citizenship). In arrows: A sufficient for B is A->B; A necessary for B is B->A

2 QUANTIFIERS & SETS

predicate - a proposition with holes: isEven(x) is not true or false until x is filled in. In code, any function returning a boolean

all (forall) - every x satisfies P; arr.every(p). True on an empty set. One counterexample kills it; no number of confirming examples proves it for an infinite set

some (exists) - at least one x satisfies P; arr.some(p). False on an empty set. One example proves it

empty-collection defaults - all([]) true, any([]) false, sum([]) 0: each returns its identity element (true for AND, false for OR, 0 for +). all(xs concat ys) = all(xs) && all(ys) must hold for empty ys, so all([]) must leave all(xs) alone

implication is a subset claim - all x in S: P(x) => Q(x) says the P-satisfiers are a subset of the Q-satisfiers. Practical (no => operator in languages): filter first - arr.filter(p).every(q)

negating quantifiers - "all X are Y" negates to "some X are not Y" (not "no X are Y"). "some X are Y" negates to "no X is Y". "all requests succeeded" negates to "at least one failed". In code: !arr.every(p) = arr.some(x => !p(x))

set - unordered, no duplicates, membership is the only question. Subset: every element of A is also in B: [...a].every(x => b.has(x))

set operations - union (in either), intersection (in both), difference A minus B (in A, not in B). Set-builder "all x in S where P(x)" is arr.filter(p). "users with access they should not have" is actual minus allowed

3 SIMPLIFYING CONDITIONALS

refactor against the truth table - list the 2^n rows and outcomes before rewriting; the rewrite must match on all rows. Catches !(a && b) "simplified" to !a && !b (differs when exactly one is true)

factoring out - (A && B) || (A && C) = A && (B || C), pull the shared term out. Most real cleanups of long conditions are this law applied once or twice

merging branches - two branches with the same body merge on OR: if (a) return f(); if (b) return f(); is if (a || b) return f()

absorption and constants - A || (A && B) = A; A && true = A; A || true = true; A && false = false. Pays off after inlining a feature flag: mechanically deletes the dead branches

exactly one of two - (A || B) && !(A && B), for booleans just A !== B. Requirements "either A or B" usually means exactly-one; code || means at-least-one - the mismatch is a requirements bug

use what the branch already knows - inside if (P), P is true; in the else branch, P is false. Substitute that in and inner conditions collapse. Swap: if P then A else B = if !P then B else A (removes a top-level NOT, mind readability)

loops are quantifiers - a scan returning true on first hit is some; returning false on first counterexample is all. for (s of servers) { if (isOffline(s)) return true } return false is servers.some(isOffline)

rewrites are only safe without side effects - if f() has effects, f() || !f() is not true, and short-circuiting means reordering a && b changes which effects run. Only rewrite pure expressions; pull effectful calls into named variables first

sets and the ability-guarantee tradeoff - a set holds less than a list (no order, no duplicates), so uniqueness is guaranteed by the type: append-if-absent becomes union. The theme: the less something can express, the more you can rely on what remains

4 PROPERTY-BASED TESTING

example vs property test - example pins one input to one output (sort([3,1,2]) is [1,2,3]); a property states a rule for all inputs ("output is ordered") and fast-check fires hundreds of random inputs at it

the four property sources - (1) output invariants: ordered, same length, same elements; (2) round trip: decode(encode(x)) = x; (3) oracle: match a slow obviously-right version; (4) relations between calls: total(cartWithItem) = total(cart) + price

shrinking - a failing random input is automatically reduced to the smallest input that still fails, plus a replay seed - the report is [0, -1], not a 40-element array

a property test is only as good as its properties - a "sort" returning the input untouched passes "same length, same elements"; only "output is ordered" kills it. Choosing properties is specification; weak properties fail silently

test strength - P is stronger than Q when P passing guarantees Q passes (P implies Q). Two ways to be stronger: more specific answer on same inputs, or same claim over more inputs - so strength is a partial order, not a ranking

property-test a refactoring - run old and new on the same generated inputs, require equal results: forall x, newF(x) = oldF(x), old code as oracle. The truth-table equivalence check automated, for when 2^n rows is too many

5 CONTRACTS & API COMPATIBILITY

precondition / postcondition - pre: what must be true at the call (caller's job); post: what is guaranteed at return (function's job). withdraw(amount): pre amount > 0 && amount <= balance; post newBalance = oldBalance - amount

contracts as asserts - preconditions asserted at the top, postconditions before return. The payoff is blame: failed pre = caller's bug, failed post = the function's own

the substitution rule - a replacement (subclass, v2, mock) may accept more (weaker pre) and promise more (stronger post), never the reverse. Liskov substitution said in contract language

backwards-compatible API evolution - v2 must accept every call v1 accepted (optional field made required = breaking) and keep every promise v1 made (removed response field = breaking). Compatible = pre only weakens, post only strengthens

type invariant - a claim true of every value of a type for its whole life, set at construction, preserved by every method: price > 0, !(available && discontinued). Contracts hold at moments; type invariants hold between them

make illegal states unrepresentable (MISU) - reshape the type so violating the invariant cannot be written: two booleans available/discontinued allow an illegal fourth combination; one status enum cannot express it

the history rule - a replacement must also change state compatibly over time: a subclass can keep every contract and invariant yet break callers by making new state sequences possible. State the sequence claim itself (append-only set: x in set implies x in set after every later step)

6 PROVING CODE CORRECT

Hoare triple - {P} code {Q}: if P holds before, Q holds after. A contract written for proving; tools like Dafny prove it for all inputs instead of asserting per run

loop invariant - true before the loop, preserved by every pass; at exit, invariant + exit condition give the result. Summing: sum equals total of arr[0..i-1] at the top of each pass; at exit i = arr.length

7 LOGIC IN DATABASES

a query is a predicate - a table is a set of facts, WHERE P returns exactly the rows where P is true, a join is an AND across two tables

NULL is unknown (three-valued logic) - any comparison with NULL is unknown, even NULL = NULL. WHERE keeps only true (drops false AND unknown); NOT unknown is unknown. x NOT IN (1, NULL) returns nothing, ever. Fixes: IS NULL, IS NOT NULL, COALESCE

constraints are invariants - UNIQUE, NOT NULL, FOREIGN KEY, CHECK are claims the database re-verifies on every write and refuses to let become false; safer there than in application code

asking for absence - "rows with no match" cannot be a plain join. Two forms: LEFT JOIN plus WHERE other.id IS NULL, or NOT EXISTS (SELECT 1 ...). Views and CTEs serve as named helper predicates

query the negation - an invariant constraints cannot express becomes a trigger: run the query finding violating rows, ROLLBACK if any exist. A state-change rule ("a set email never returns to NULL") is a BEFORE UPDATE trigger whose WHEN clause is the rule's if-part

8 DECISION TABLES

decision table - one column per condition, one row per combination, one outcome per row; n booleans = exactly 2^n rows (enumeration columns fine if values cover the domain). FizzBuzz is a 4-row table on n%3 and n%5

the consistency rule - every possible input maps to exactly one row: a combination no row covers is a gap (never decided); two rows covering the same inputs with different outcomes is an overlap, a contradiction. Two rows may share an outcome. Enumerate, then ask the product owner one blank cell at a time

don't-care, irrelevance, decomposition - mark ignored conditions - and merge rows. Cleanups: a column that never changes any outcome gets deleted (irrelevance); some tables split into two independent smaller ones (decomposition). One test per row = complete coverage; generate the suite from the table

9 SOLVERS

the declarative flip - imperative code says how to build the answer; a solver takes what a valid answer looks like (variables, allowed values, constraints) and searches. Use when rules are easy to state and the procedure is hard

constraint solving (the suitcase) - items with weight and value, capacity 10kg: one yes/no variable per item, weights sum <= 10, maximize value sum. Tools: MiniZinc, OR-Tools. Same shape: scheduling, seat assignment

SAT and SMT - SAT: find a true/false assignment satisfying a boolean formula. SMT = SAT + theories (integers, arrays) so x + y < 10 is stated directly; Z3 is the standard. Three answers, not two: sat, unsat, unknown (a real answer code must handle). Runs today in package version resolution

satisfaction vs optimization - satisfaction: any assignment meeting the constraints; optimization: add a cost, solve minimize. Fleet example: server counts, total requests >= 7500, minimize cost. Symmetry breaking: pick one representative of interchangeable solutions so the solver skips twins

proving with a solver - hand the solver the claim's negation: unsat = no counterexample = proved; sat = concrete counterexample. Z3: a >= b, check not (a*c >= b*c) - sat with negative c; add c >= 0, unsat, proved. Same move checks code against postconditions: a Hoare triple verified by search

10 LOGIC PROGRAMMING

Prolog - facts (parent(a1, a2). - commit a1 is parent of a2) and rules (ancestor(A, C) :- parent(A, C). and ancestor(A, C) :- parent(A, Y), ancestor(Y, C). where :- reads "if"); a query searches for every derivable answer (ancestor(X, a4), merge commits = two distinct parents), and the same rule runs in any direction

Datalog - Prolog restricted so every query terminates (ability-guarantee tradeoff); the query language inside static analyzers (CodeQL) and Datomic. Deductive-database read: high-churn files, commits missing test changes - two lines of rules each

answer set programming - clingo: choice rules generate candidates (each task assigned to exactly one worker), a bare ":- body" is an integrity constraint (answers where body holds are thrown out), #minimize picks the best answer. Same family (Picat) does planning: actions as state rewrites with costs, cheapest sequence from start to goal