

EXAM DNA REPORT

CS 61A

Exam DNA Lite — Free Edition

The archetype map: how this exam thinks, where the points concentrate, and the traps that actually cost grades.

Edition: Fall 2026 · v1.0

CA

HOW TO USE THIS REPORT

1. Read §2 (the Archetype Map) before anything else — it tells you what the exam actually asks. **2.** Run §7's 10-day order the two weeks before your exam. **3.** Bring §4's traps to your practice sessions and check each one off as you de-fang it. **4.** Pair everything with the course's own public exams (§9) — this report tells you how to use them, not what they say.

This report is original analysis of how the course's exams are *structured*. It quotes no exam questions. Verify specifics against your current syllabus each semester — courses drift.

1 • The Exam Shape

- **Format:** In most recent formats, midterms and the final are closed-book, closed-computer paper exams — you write Python (and Scheme) by hand, with a printed reference sheet of selected built-ins provided.
- **Timing pressure:** Roughly one substantial multi-part problem every 15–20 minutes. Most students cannot attempt every sub-part; triage is a graded skill in itself.
- **Question style:** Long scenario-driven problems — a small program or environment is described, then you trace it, extend it, or implement a missing piece. Environment diagrams, recursive function design, and Scheme evaluation appear in nearly every sitting.
- **What the exam rewards:** Precision over cleverness. Exact return values, exact frames and bindings, exact base cases. Partial correctness earns little; a fully correct small function earns a lot.
- **What it punishes:** Vague pseudocode, "almost right" recursion, and misreading which function calls which — the reason §3's read-every-sub-part-twice rule exists.

2 • The Archetype Map

Archetype	What it looks like	Typical weight	Difficulty ceiling
The Frame Autopsy	Draw a complete environment diagram for a tricky nested-call program with lambdas, reassignments, and higher-order functions	High on every exam	Brutal — one wrong arrow cascades
The Recursive Sculptor	Implement a recursive function over trees, linked lists, or numbers where the base case is subtly non-obvious	High	High — the "obvious" base case is usually wrong
The Function Factory	Write a higher-order function that returns, composes, or parameterizes other functions	High	Moderate-to-high
The Scheme Interpreter's Nightmare	Evaluate or hand-trace a Scheme expression with <code>let</code> , <code>define</code> , tail calls, or streams	Medium	Moderate — mechanical but detail-hungry
The Iterator Whisperer	Predict behavior of generators, <code>iter/next</code> , and lazy sequences, or implement one	Medium	High — laziness semantics trip everyone
The Growth Curve	Big-O analysis of a given function or loop nest, sometimes with a twist that changes the naive answer	Medium	Moderate

Archetype	What it looks like	Typical weight	Difficulty ceiling
The Data Wall	Implement or extend an object-oriented class hierarchy with inheritance, properties, and dunder methods	Medium	Moderate
The One-Liner Gauntlet	Short "fill in the blank" expressions using map, filter, min/max with keys, or list comprehensions	Medium	Low-to-moderate — cheap points if fluent
The Mystery Trace	Given unfamiliar code, state what it prints or returns, often involving exceptions or non-obvious control flow	Medium	Moderate
The SQL Interlude	Write a query or predict query output over small tables (in formats that include SQL)	Low-to-medium	Moderate
The Tail-Call Conversion	Rewrite a recursive Scheme procedure so it runs in constant space	Low	Moderate
The Debugging Bench	Identify and fix a specific bug in provided code, then justify the fix	Low-to-medium	Low — very bankable

3 • The Point Economy

Points concentrate in a small number of heavyweight problems — usually one environment diagram, one or two recursive implementations, and one higher-order design problem. These are worth more individually than the entire short-answer section, but they are also where most students bleed time. The short-expression and trace questions at the front are the "bank": fast, deterministic, and forgiving. The dangerous zone is the last third of the exam, where a hard recursive problem sits next to an unfamiliar scenario; students who grind on the hard problem's first sub-part often abandon easy sub-parts worth comparable points behind it.

- **Bankable:** one-liner expressions, Scheme evaluation, straightforward traces, debugging fixes.
- **Bankable with practice:** environment diagrams — they follow a strict mechanical procedure.
- **Dangerous:** open-ended recursive design with an unusual data shape; generator semantics under mutation.
- **Dangerous:** any sub-part whose wording you haven't read twice — misread prompts are the top silent point-killer.
- **Strategic rule:** in most recent formats, securing every bankable point plus half of each hard problem beats fully solving one hard problem and skipping the bank.

4 • Trap Patterns

- **The rebinding trap.** A name is reassigned after a lambda or function is defined; students evaluate with the old binding. → *Counter:* every environment diagram question is testing when names are *looked up*, not when they're defined — always resolve at call time.
- **The default-argument snapshot.** A mutable default argument is shared across calls. → *Counter:* memorize that defaults are evaluated once, at definition; annotate any default that's a list or dict.
- **The base-case mirage.** A recursive function looks like it needs a base case for the empty input, but the recursion actually bottoms out one level earlier (or later). → *Counter:* hand-run the smallest two inputs before writing anything.
- **The Scheme quoting trap.** Confusing a symbol, a quoted list, and an evaluated expression in define/let forms. → *Counter:* ask "is this evaluated?" before every subexpression; quote marks

are load-bearing.

- **The generator exhaustion trap.** Iterating a generator twice and expecting the second pass to yield values. → *Counter:* generators are one-shot; check whether the problem re-iterates anything.
- **The inheritance lookup trap.** Predicting which class's attribute or method is found when multiple levels define it. → *Counter:* trace the lookup order explicitly, bottom-up, before answering.
- **The truthiness trap.** Using a value where a predicate is expected (or vice versa) in filter/min/and/or chains. → *Counter:* write out what each operand evaluates to, not just whether it's "truthy."
- **The tail-call trap.** Assuming a recursive call inside an expression (like inside +) is a tail call. → *Counter:* a call is a tail call only if nothing happens after it returns.
- **The mutation aliasing trap.** Two names referring to the same list, mutated through one. → *Counter:* draw the box-and-pointer picture; never reason about lists by name alone.

This is the free lite edition — sections 1–4 of 9. The full Exam DNA Report adds grader psychology (§6), the 10-day rehearsal order (§7), AI failure modes with 2-minute verification checks (§8), and the curated Free Stack (§9). Get it at clarityacademic.com/courses/cs61a.html

© **Clarity Academic** · Original analysis — not affiliated with, endorsed by, or sponsored by UC Berkeley.