

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.

MOCK EXAM A — Frames, Recursion, and Functions as Values

CS 61A · Mock Exam A — Frames, Recursion, and Functions as Values 100 points · 110 minutes
· closed book, no computer · 7 problems

Permitted materials: the course's printed reference sheet of selected built-ins — the sheet the hall provides in most recent sittings (DNA §1); you bring nothing. Nothing else on the desk. Original practice exam — not affiliated with UC Berkeley. Not drawn from past exam questions.

Problem 1 (10 points). *Archetype: the One-Liner Gauntlet (DNA §2).* Write the exact value each expression evaluates to, then one sentence naming the rule that decides it. Each sub-part is worth 1 point for the value and 1 for the rule.

(a) (2 points) `min(["rake", "hoe", "saw"], key=len)`

(b) (2 points) `0 or [] or 'silt' and 7`

(c) (2 points) `[x for x in [0, '', 'silt', 3, None, [], [0]] if x]`

(d) (2 points) `list(map(lambda p: p[0] * p[1], [(2, 5), (3, -1), (0, 9)]))`

(e) (2 points) `sorted([[2, 3], [1, 9], [2, 1], [1, 2]], key=lambda p: (p[0], -p[1]))`

Problem 2 (10 points). *Archetype: the Mystery Trace (DNA §2).*

```
def fold_in(pairs):
    result = []
    index = 0
    for k, v in pairs:
        if index % 2 == 0:
            result.append(k + v)
        else:
            try:
                result.append(result[index - 1] // v)
            except ZeroDivisionError:
                result.append(index)
        index += 1
    return result
```

(a) (4 points) Write the exact list that `fold_in([(4, 6), (10, 0), (1, 5), (7, 2), (0, 0)])` returns.

(b) (4 points) Recompute the same call assuming the except branch appended `v` instead of `index`. State which positions change and why.

(c) (2 points) For `fold_in([(2, 'a'), (3, 1)])`: state which expression raises during the first iteration, which exception it raises, and whether either except branch in `fold_in` catches it.

Problem 3 (8 points). *Archetype: the Growth Curve (DNA §2).* State each bound as $\Theta(\dots)$ in terms of n with the justification the sub-part asks for.

(a) (2 points)

```
def grid(n):
    count = 0
    i = 0
    while i < n:
        j = 0
        while j < n:
            count += 1
            j += 1
        i += 1
    return count
```

State Θ for `grid(n)`; justify in one sentence.

(b) (4 points)

```
def sweep(n):
    count = 0
    i = 1
    while i <= n:
        j = i
        while j <= n:
            count += 1
            j = 2 * j
        i = i + 1
    return count
```

State Θ for `sweep(n)`; justify by grouping iterations by inner-loop length. The naive first reading is wrong here — say why in one sentence.

(c) (2 points)

```
def carve(n):
    if n <= 1:
        return 1
    return carve(n // 2) + carve(n // 2) + n
```

State Θ for `carve(n)`; justify with the recursion tree (work per level times number of levels).

Problem 4 (20 points). *Archetype: the Frame Autopsy (DNA §2).*

```
scale = 10

def pack(item, trail=[]):
    trail.append(item)
    def show():
        return str(scale) + ':' + str(len(trail))
    return show

pack('lamp')
s1 = pack('rim')
s2 = pack('urn', [])
scale = 100
print(s1(), s2())
```

(a) (8 points) Draw the complete environment diagram as of the moment the `print` line begins: every frame created by the calls above — including the discarded first call — parent pointers on every

THE DECK, TOPIC BY TOPIC

1 • The Frame Autopsy (28 cards)

Card	Answer
In an environment-diagram problem, what is the complete graded artifact?	Every frame with its parent arrow, every binding, and each call's return value. One wrong arrow cascades through every later lookup — completeness is the point. Source: DNA §2.1 — what a Frame Autopsy demands
Python calls a function — what two things does the new frame record immediately?	The parameters, each bound to its argument's value in order, and the parent pointer — set to the frame where the function was DEFINED. Source: DNA §5.1 — frame creation
A name is evaluated inside a function body — where does lookup search, in order?	The current frame, then each parent frame outward, then the global frame, then built-ins — stopping at the first frame that holds the name. Source: DNA §5.1 — the lookup order
What exactly does a function value's parent pointer point to?	The frame where that function was defined — for a lambda, the frame where the lambda was evaluated; for a def, the frame where the def statement ran. Source: DNA §5.1 — the parent-pointer rule
A def statement executes inside a frame — what binding does it create?	The def'd name is bound in the current frame to a function whose parent is that same frame — the def itself creates no frame. Source: DNA §5.1 — how def binds
Does evaluating a lambda expression create a frame?	No — it evaluates to a function value (parameters, body, parent). A frame appears only when that function is called. Source: DNA §2.1 — lambdas in the diagram
A function is called from a different frame than the one it was defined in — which frames can its body's lookups reach?	Only its defining environment, via the parent pointer — never the caller's frames. Parents follow definition, not the call site. Source: DNA §5.1 — lexical scoping
A name exists in both an enclosing frame and the global frame — which binding wins?	The nearest one along the parent chain — lookup moves outward from the local frame and stops at the first hit. Source: DNA §5.1 — the shadowing rule
An assignment rebinds a name the frame already binds — what changes in the diagram?	The existing row's value is overwritten — one row per name per frame. Every later lookup through that frame sees the new value. Source: DNA §4.1 — the rebinding mechanic
A lambda is defined; a name it references is reassigned; only then is the lambda called. What does its body see?	The new binding. The rebinding trap: names resolve at look-up time, not at definition time — always resolve at call time. Source: DNA §4.1 — the rebinding trap
f(g) versus f(g()) — what is the diagram-level difference?	f(g) passes the function object itself — no call, no frame. f(g()) calls g first: draw g's frame, and its return value becomes f's argument. Source: DNA §2.1 — function value versus call
The same function is called twice in one run — how many frames?	Two — every call gets its own frame with its own parameter bindings; the calls share nothing except ancestors. Source: DNA §2.1 — one frame per call
A lookup consults "the environment" — what is that, concretely?	The chain of frames reachable by following parent pointers from the current frame — the frames a lookup consults, local first, ending at global. Source: DNA §5.1 — the environment definition
In a call expression f(a, b), what is evaluated and in what order?	The operator first, then each operand left to right — either may spawn its own frame — then the resulting function is applied to the resulting arguments in a new frame. Source: DNA §2.1 — evaluation order
Where does a finished call's return value appear in the diagram?	At the call site — recorded as the return value of the call expression in the calling frame; the callee's frame then stops being consulted. Source: DNA §2.1 — return values in the diagram

THE 14 DAYS

Days 1–10 embed the DNA §7 order 1:1 — archetype, sequence, and dose logic unchanged; each day adds one escalation line. Days 11–14 are the sprint's mock cycle; Part 3 defines every selection rule they reference. Read each day as its four lines: the archetype cluster, the dose, the verifiable output, the escalation.

Dose fidelity: Days 1–8 keep the DNA §7 counts and caps as written — 10 diagrams; 20 plus 10 at two minutes each; 8; 5 at fifteen minutes; 6; 15 plus 2; 4 plus 8; 15 plus 6 at the 90-minute cap. The escalation line adds; it never swaps the dose. Days 9 and 10 are the DNA §7 rehearsal and autopsy slots, instantiated with the pack's Mock A. Days 11–14 are days the DNA does not schedule — the sprint's additions — and their selection rules come from Part 3.4.

Failed-output rule: a day that ends short of its output carries the remainder to the next morning's first 30 minutes, logged on the window map like any displaced dose (Part 1.4). A skipped day is not a collapsed plan; it is a displaced dose.

- **Day 1 — Archetype(s):** the Frame Autopsy.
 - **Dose:** 10 environment diagrams self-drawn from past-exam code in the course's public archive (DNA §9), until the arrow procedure is mechanical. Cap 120 minutes.
 - **Output:** 10 completed diagrams with parent pointers on every frame and function value; a lookup log per diagram — each name, the frame it resolved in, the moment it resolved (call time).
 - **Escalation:** weaponize the rebinding trap (§4) — after each diagram, redraw only the lookup arrows against a 20-seconds-per-lookup clock. Any lookup you resolved at definition time is a rep to redo. This is §5 level 1 (names, frames, environments) drilled at speed; every other archetype stands on it.
- **Day 2 — Archetype(s):** the One-Liner Gauntlet, the Mystery Trace.
 - **Dose:** 20 one-liner expressions and 10 mystery traces, timed at two minutes each. Cap 120 minutes.
 - **Output:** 30 paper answers with per-item times; a correct-of-30 tally; every miss annotated with the one rule that decides the item.
 - **Escalation:** weaponize the truthiness trap (§4) — on every `and / or / filter` item, write what each operand evaluates to before answering. This cluster is the exam's cheap-points bank (§3), so §5 level 2 (functions as values) drills against the two-minute clock: the clock is the drill; a correct untimed answer scores zero under exam pacing.
- **Day 3 — Archetype(s):** the Recursive Sculptor, easy tier.
 - **Dose:** 8 base-case-first problems over numbers and lists. Cap 120 minutes.
 - **Output:** 8 functions, each written base-case-first; the two smallest inputs hand-run per problem, values written down before the recursive case.
 - **Escalation:** weaponize the base-case mirage (§4) — the smallest-two-inputs pass runs under two minutes per problem, clock on, before anything else gets written. §5 level 3: base case first, then trust the recursive leap.
- **Day 4 — Archetype(s):** the Recursive Sculptor, hard tier.
 - **Dose:** 5 tree and linked-list problems, timed at 15 minutes each.
 - **Output:** 5 solutions inside the cap; the base case written first on every problem; a box-and-pointer picture for every Link item.