Clojure in Academia

Teaching Functional Programming to Students Who Aren't Looking For It

Jim E. Newton
EPITA Research Laboratory

Abstract

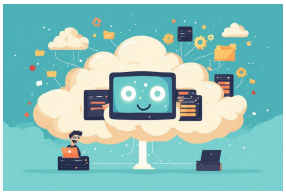
This article is a companion to a talk of the same name given at clojure.conj 2026 (September 30 – October 2, 2026, Charlotte, NC, USA). It exists for anyone who wants the substance of the talk without having attended it, and it goes into more depth than the slides could hold. At EPITA (Paris) we teach an elective on functional programming using Clojure. Most students don't fully "get it" by the end. A few do, and the difference is striking. This article is a candid look at that divide: which ideas consistently fail to land, what lands for the minority who connect, how the constraints of a real course shape the outcome, and what three years of submission data do and do not tell us about whether students are doing their own work. Nothing here is a rigorous study. It is an honest account of one small, real course.

1 Introduction

This account comes out of EPITA (Paris), where we have taught this functional-programming elective – first in Scala, now in Clojure – for three years. Our own programming background spans roughly 40 years, mostly in Lisp dialects: Emacs Lisp, Common Lisp, SKILL++, and Clojure for about the last 8 of those years. What follows is a report of our own experience and findings from that time, not a general claim about teaching functional programming.

This abstract for the talk this article accompanies was written months before the course it describes actually ran. It promised a story about immutability, recursion, and lazy evaluation. What actually turned out to matter more was more mundane: attendance economics, students quietly deciding how much effort a given grade is worth, and a pattern in three years of submission data that we still cannot fully explain. We think that is a more honest article than the one we originally set out to write, and we would rather tell you that up front than let you discover it halfway through.

2 The Course



The course is an elective at EPITA (Paris): *Techniques in Functional Programming*. It was ported from an existing Scala elective, *Applications in Functional Programming* — the premise in both is the same: take algorithms students already know, and rewrite them in a functional style. This offering used Clojure instead of Scala.

The course runs 4 three-hour sessions in a single semester. It is mostly live coding: we explain as we go, deliberately create errors live, and ask students to help figure out what went wrong. Sessions 2, 3, and 4 each begin with a short, intentionally easy quiz.

Session by session: the first covers language overview, syntax, and recursion – plain recursion, tail recursion, and `loop`/`recur` – along with multiple-arity `reduce` and a first look at multi-threading. The second covers higher-order functions and immutability, using a graph-theory exercise (building an adjacency list) and the sequence functions `map`, `mapcat`, `filter`, and

`group-by` over vectors, lists, sets, and maps. The third is built around the `for` loop, applied to CSV parsing and plot generation with Vega. The fourth covers concurrency – futures and atoms – with examples contributed by a student, `@andrey.fadeev`, from an earlier year.

2.1 Teaching Philosophy

Our approach rests on a few simple assumptions: that students actually want to learn the material, so class time should not be spent on busywork; that it helps to think like a beginner rather than an expert; and that showing only the efficient, polished solution hides more than it teaches. So we make mistakes live, on purpose, and let the class watch them get found and fixed, rather than presenting code that already works.

Reception to this is mixed. It works well for some students; others, particularly those taking careful notes, dislike having errors recorded in their own notebooks. Doing it well is also harder than it sounds inside a three-hour session: we learn by doing things wrong and only afterward understanding why they were wrong, and that understanding takes time a tight classroom budget does not always have.

2.2 Why Aren't They Looking For It?

Calling it "an elective" undersells how little some of these students actually chose functional programming specifically. Three reasons are worth spelling out.

First, not all of them are electing anything: the same course, first in Scala and now in Clojure, has run at different times as both an elective and a required course. Students in the required version are, by definition, not there because they went looking for functional programming.

Second, most courses at EPITA are taught in French; the handful taught in English exist partly to serve exchange students who do not speak French. Those students pick from whatever English-taught electives are on offer that semester, regardless of subject – functional programming is, for some of them, simply the one that fit their schedule and their language.

Third, even among students who genuinely chose the course for its subject matter, the advertised title – *Techniques in Functional Programming* – never mentions Clojure, or Lisp, or parentheses at all. A student can walk in wanting functional programming and still not have been looking for *this*.

2.3 Grading

The final grade is Quiz 1 (10%) + Quiz 2 (10%) + Quiz 3 (20%) + Homework (60%). Homework is the average percentage of tests passed across 12 required assignments, submitted by pushing a git tag (`<assignment-slug>-<suffix>`) to a per-student grading repository. The push triggers MAAS – *Moulinette as a Service* — EPITA's automatic grading pipeline. A *moulinette* is, literally, a small mill or grinder, normally hand-operated — the kind used in a French kitchen to purée vegetables by turning a crank. In French computer-science slang the word has long meant a script that "grinds" a student's submission through a test suite to produce a grade; historically that script was something a teaching assistant ran by hand. MAAS is the same idea, just running automatically, on a server, for every push. It grades against a server-side *golden* copy of the test suite — students can see the tests and edit their local copy freely, but editing it has no effect on the grade, since grading always runs against the golden copy. Every required assignment counts the same, regardless of how hard it is.

Students normally test everything on their own laptop first and only push once it already passes there — but the official grade always comes from MAAS, not the laptop, and the two environments are not identical. The cloud runner has a fixed, fairly modest memory ceiling and always runs Linux, so a student on Windows, or on a laptop with far more memory than the grader allows, can occasionally see code behave differently once it actually reaches MAAS, for reasons that have nothing to do with whether the solution itself is correct.

Students are told explicitly: you do not need 100%. Decide what grade you want, and do enough problems — factoring in your quiz score — to get

there.

After every push, the student can use the web page shown alongside this paragraph to see which tests passed and which failed, along with the actual error messages and stack traces — the same feedback we would see debugging it ourselves, not just a bare pass/fail count.

2.4 Why Quizzes Are 40% of the Grade

Two reasons, one pedagogical and one just honest.

The homework score does not capture the whole picture. Passing tests measures whether code runs; it does not measure what a student picks up from being in the room — questions asked, mistakes debugged live, another student's confusion becoming your own moment of clarity. Section 5 shows what that divide looks like in the actual gradebook.

And, more selfishly: we prepare for every lecture. For an elective, it is not unusual for only one or two students to show up to a room meant for twenty. That is genuinely hard to keep doing. Weighting the quiz rewards showing up — partly for the students' benefit, and partly for mine.

3 The Assignments

Thirteen assignments are required: `hello`, `recursion`, `reduce`, `factors`, `theg`, `polynomial`, `binarysearch`, `geodesic`, `cakecutting`, `trig`, `runaverage`, `frenchnames`, and `xlookup`. Three more (`calculus`, `determinant`, `triptych`) are given as templates but are not graded.

Each assignment is a template file: a docstring specifying the contract, a stubbed-out function body, and a provided, non-secret test file. Listing 1 shows the smallest one in full.

```
(defn hello
  "Given who, e.g. \"Jim\", returns
   \"Hello, Jim.\""
  [who]
  (throw (ex-info "not yet implemented" {})))
```

Listing 1: *The `hello` assignment — the simplest of the thirteen, but not merely a tooling check.*

3.1 What Each Assignment Asks For

For anyone who wants more than the two-example taste given in the talk itself, here is what each of the thirteen required assignments actually asks a student to build.

Below, each assignment's description is followed by the actual template for what we judged to be its main function — the stubbed-out code exactly as a student receives it, omitting supporting functions where the assignment has more than one. Some of these are long; that is itself part of the point made in Section 3.2.

`hello` The simplest assignment, but not merely a tooling check, and it counts exactly as much toward the grade as any other — every required assignment is weighted equally, regardless of difficulty. It tests real basic syntax competency (function definition, s-expressions, string literals), and it also requires looking up the documentation of one or more Clojure functions to build the result string, which is arguably the point: confirming a student can write and submit working Clojure at all, before anything harder is asked. Shown in full above in Listing 1.

`recursion` Sum and product of a list of numbers, implemented three parallel ways: plain recursion, `loop`/ `recur`, and `reduce`. Foundational idiom practice, not a puzzle. The `reduce` variant of sum:

```
(defn sum-by-reduce
  "Use reduce to sum the elements in a given list"
  [nums]
  (reduce (throw (ex-info "Missing single expression, not yet implemented" {}))
          (throw (ex-info "Missing single expression, not yet implemented" {}))
          ))
```

Listing 2: *`recursion`: `sum-by-reduce`.*

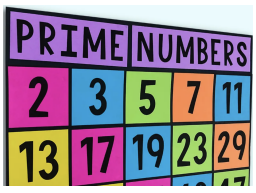
`reduce` A harmonic-sum and a running-sum computed with `reduce`, plus variadic addition and multiplication over pairs of numbers (modeling complex-number arithmetic). The pair-arithmetic identity elements are never stated — see Section 3.2. The multiplication side of that puzzle:

```clojure
(defn times
  "e.g.  (times [3.0 5.0] [-7.0 2.0]) => [-31.0 -29.0]
         (times) => identity for multiplication
         (times a) => a
         (times ...) => product of a sequence of values"
  ;; The value which we can multiply (times) to any (Double Double)
  ;;    and get the same value back.
  ;;    (times times-identity  x) == x
  ;; and (times x  times-identity) == x
  ;; If it is not clear what this value should be
  ;; you may look closely at the test cases;
  ;; you should be able to reverse engineer the value of
  ;; times_identity by looking at the test case.
  ([] (throw (ex-info "Missing single expression, not yet implemented" {}))
   )
  ([a] a)
  ([[a b] [c d]]
   [(- (* a c) (* b d))
    (+ (* b c) (* a d))])
  ([a b & rest]
   ;; You may use your function `times` to compute the product
   ;; of exactly two pairs at a time.
   ;; Use reduce to compute to generalize to a List of pairs of Double
   ;; You'll need to provide a so-called zero argument for fold in both
   ;; cases. In the first case the argument must be the identity of the
   ;; plus function  and in the second case  you'll need the identity
   ;; for the times function.   You need to figure out what these
   ;; elements are?  E.g.  which pair [e1 e2] has the property that
   ;; for all [a b]  (times [e1 e2] [a b]) == [a b] == (times [a b] [e1 e2]) ?"
   (throw (ex-info "Missing single expression, not yet implemented" {}))
   ))
```

Listing 3: *`reduce`: `times` — the identity element is never stated.*

`factors` Prime factorization of a whole number by trial division. The algorithm is familiar to nearly everyone; getting a correct, reasonably efficient loop written is the actual work. The entire assignment is this one function — the hint alone (45 lines) is longer than any other required assignment's:

```clojure
(defn prime-factors
  "Return a vector of positive integers, each >= 2,
  representing the prime factorization of the given whole number.
  The order of the prime factors is not important.
  The product of the factors must be n, and each of the
  factors must be prime."
  [n]
  (assert (> n 0))
  (assert (integer? n))
  ;; CHALLENGE: student must complete the implementation.
  (throw (ex-info "Missing one or more expressions, not yet
      implemented" {}))
  ;; HINT 45 line(s)
)
```



Listing 4: *`factors`: `prime-factors`.*

One possible completed solution looks like this:

```clojure
(defn prime-factors
  "Return a vector of positive integers, each >= 2,
  representing the prime factorization of the given whole number.
  The order of the prime factors is not important.
  The product of the factors must be n, and each of the
  factors must be prime."
  [n]
  (assert (> n 0))
  (assert (integer? n))
  (if (even? n)
    ;; if n is even, the prime factors are 2 followed by the factors of n/2
    (conj (prime-factors (/ n 2)) 2)
    ;; now we know n is odd, so when we iterate, we can skip even candidates.
    (loop [primes []
           n      n
           start  3                    ;; lower limit
           ul     (round (sqrt n))]    ;; upper limit
      (cond (= 1 n) primes

            (< ul start)
            (conj primes (int n))

            (zero? (mod n start))
            (recur (conj primes start)
                   (/ n start)
                   start
                   (round (sqrt (/ n start))))

            :else
            (recur primes n (+ start 2) ul)))))
```

Listing 5: *`factors`: one working `prime-factors` solution.*

Each of the three places the stub leaves blank rests on a fact worth stating precisely rather than taking on faith — the same four theorems (with the same proofs) that justify this exact algorithm in the Algebra course's own recursion chapter, which teaches `prime_factors` in Python before any student sees this Clojure version.

Theorem 1 (Smallest Factor Is Prime)**.** *If n is composite, its smallest factor greater than* 1 *is prime.*

Proof. Suppose p is the smallest factor of n, with $n = pk$, and suppose toward contradiction that p itself is composite, say $p = ab$ with $1 < a < p$. Then $n = abk$, so a is a factor of n smaller than p — contradicting that p was smallest. So p must be prime. □

This is why the loop never has to separately check that `start` is prime once it finds `(zero? (mod n start))`: the first divisor ever found, searched in increasing order starting from the smallest candidate, is automatically prime.

Theorem 2 (Smallest Factor Is Bounded)**.** *If n is composite, its smallest factor is at most $\sqrt{n}$.*

Proof. Suppose otherwise: n's smallest factor p satisfies $p > \sqrt{n}$, and $n = pq$. Since p is smallest, $p \leq q$, so both exceed $\sqrt{n}$; write $p = \sqrt{n}+\alpha$, $q = \sqrt{n}+\beta$ for some $\alpha, \beta > 0$. Then $n = pq = n+\sqrt{n}(\alpha+\beta)+\alpha\beta > n$, a contradiction. □

This is exactly why the `(< ul start)` line is safe: once `start` exceeds `ul` (the rounded $\sqrt{n}$) without a factor ever being found, n cannot be composite — it must be prime, and `(conj primes (int n))` is the correct final answer.

Theorem 3 (Dividing Out Preserves the Lower Bound)**.** *If s is the smallest factor of n, then n/s has no factor smaller than s.*

Proof. Suppose otherwise: some $q < s$ divides n/s, and $n/s = qr$ for some r, so $n = qrs$. Since q and s are both prime (by the previous theorem) and $q < s$, this makes q a factor of n smaller than s — contradicting that s was n's smallest factor. □

This is why the recursive step for a genuine divisor keeps `start` unchanged rather than resetting it to 3: having found that `start` divides `n`, we already know the remainder `(/ n start)` has no smaller factor left to search for, so searching again from `start` — which correctly catches repeated factors like the 2's in $8 = 2^3$ — is not just convenient, it is provably complete.

Theorem 4 (Odd Factors)**.** *If n is odd, every factor of n is odd.*

Proof by contrapositive. If n has an even factor $2p$, then $n = 2p \cdot q$ for some q, so n is even. □

By the time the loop runs, n has already had every factor of 2 stripped out by the `even?` branch above, so it is odd — which is why the `:else` branch advances `start` by 2 rather than 1: every even candidate is guaranteed not to divide an odd number, so testing it would be wasted work.

One case is worth tracing by hand, because it is easy to worry the algorithm gets it wrong: a repeated prime factor sitting exactly at the boundary, where `start` and `ul` coincide.

Example 1 (A Repeated Factor Exactly at the Boundary)**.** Take $n = 17^2 = 289$.

The loop begins with `primes=[]`, `n=289`, `start=3`, `ul=(round (sqrt 289))=17`. Stepping through the odd candidates $3, 5, 7, 9, 11, 13, 15$ finds no divisor, until `start` reaches 17 — the same value as `ul`. At that point `(< ul start)` is `(< 17 17)`, which is *false*: the check is strict, so reaching the boundary does not give up the search. `(zero? (mod 289 17))` is true, so the loop recurs with `primes=[17]`, `n=(/ 289 17)=17`, `start=17` (*unchanged*), and a freshly computed `ul=(round (sqrt 17))=4`.

On this next pass, `n=17`, `start=17`, `ul=4`, and now `(< ul start)` is `(< 4 17)`, which is true: the remaining 17 is concluded to be prime, and the loop returns `(conj [17] 17) = [17 17]` — correct.

Two details make this come out right, and neither is an accident. First, the strict `<` in `(< ul start)` is the code-level expression of Theorem 2's *at most* $\sqrt{n}$: since the bound is inclusive, the give-up check must fire only once `start` has gone strictly past `ul`, never when they are equal. Second, the recursive call keeps `start` unchanged rather than advancing it, which is exactly what lets the same prime be found a second time — the code-level expression of Theorem 3.

`theg` Graph theory. Build an adjacency-list representation from a list of edges using `group-by`, then implement a breadth-first reachable-vertices search and a distance-from-source partitioning, both via an explicit `loop`/`recur`:

```
(defn reachable-vertices
  "Return a set of all verticies which are reachable starting at vertex `v-start`.
  The graph can be directed or undirected depending on the input parameter `directed`"
  [edges v-start directed]
  (let [adj (make-adj edges directed)]
    (loop [done #{}
           to-do #{v-start}]
      ;; CHALLENGE: student must complete the implementation.
      (throw (ex-info "Missing one or more expressions, not yet implemented" {}))
      ;; HINT 7 line(s)
      )))
```

Listing 6: *`theg`: `reachable-vertices`.*

`polynomial` Represent a polynomial as a map from exponent to coefficient, and implement its algebra: addition, subtraction, scalar scaling, multiplication (explicitly flagged in the assignment as "not trivial" — every term of one polynomial must be multiplied against every term of the other), exponentiation by repeated squaring, polynomial long division, and two flavors of equality. An earlier version of this assignment also required implementing root-finding (quadratic formula, bisection, and a case-based recursive solver for higher degrees); that part has since been removed as an unnecessary complication once it became clear it was not actually required of students. The function flagged as not trivial:

```
(defn poly-times
  "multiply two polynomials.
  You may find this function challenging to write.  It is not
  trivial.   Remember that every term in a must be multiplied by every
  term in b."
  [a b]
  ;; CHALLENGE: student must complete the implementation.
  (throw (ex-info "Missing one or more expressions, not yet implemented" {}))
  ;; HINT 9 line(s)
  )
```

Listing 7: *`polynomial`: `poly-times`.*

`binarysearch` A generalized binary search over an arbitrary boolean step function, including automatically finding a bracketing interval by doubling outward before bisecting — a more abstract form of "binary search" than the sorted-array version most programmers learn first. Again, the entire assignment is one function:

```clojure
(defn bin-search-by-boolean
  ([f-step delta]
   ;; the 2-argument clause computes left and right and calls the
   ;; 5-argument entry point with max-depth=false
   (loop [left -1.0
          right 1.0]
     ;; CHALLENGE: student must complete the implementation.
     (throw (ex-info "Missing one or more expressions, not yet implemented" {}))
     ;; HINT 5 line(s)
     ))
  ([left right f-step delta max-depth]
   (assert (<= left right))
   ;; takes a function, f, for which f(left) = false, and f(right) = true,
   ;; finds an x such that f(x)=false, and f(x+delta)= true

   (case (list (boolean (f-step left))
               (boolean (f-step right)))
     ((true true) (false false))
     false

     ((true false))
     ;; complement the function and make recursive call
     (bin-search-by-boolean left right
                            (throw (ex-info "Missing single expression, not yet
                                implemented" {}))
                            delta max-depth)

     ((false true))
     (loop [left  left
            right right
            depth 0]
       (let [mid (throw (ex-info "Missing single expression, not yet implemented" {}))
             ]
         (cond (< (- right left) delta)
               (throw (ex-info "Missing single expression, not yet implemented" {}))

               (and max-depth  ;; check max-depth as necessary
                    (>= depth max-depth))
               (throw (ex-info "Missing single expression, not yet implemented" {}))

               (f-step mid)
               (throw (ex-info "Missing single expression, not yet implemented" {}))

               :else
               (throw (ex-info "Missing single expression, not yet implemented" {}))
               )))

     (throw (ex-info "line never reached" {})))))
```

Listing 8: *binarysearch: bin-search-by-boolean, the whole assignment.*

geodesic Geometry on a sphere: great-circle distance between two (latitude, longitude) points, conversion to and from 3-D coordinates, correctly normalizing coordinates that wrap past the poles or the antimeridian, bisecting a geodesic segment, and tracing a full geodesic as a sequence of closely-spaced points. See Section 3.2. The formula-lookup function itself:

```clojure
(defn geodesic-distance
  "km distance between two [lat lon] points"
  [[lat-1 lon-1] [lat-2 lon-2]]
  (let [phi-1 (to-radians lat-1)
        phi-2 (to-radians lat-2)
        lambda-1 (to-radians lon-1)
        lambda-2 (to-radians lon-2) ]
    (* earth-radius
       ;; The student should find the mathematical formula for
       ;; distance over the surface of the globe between two
       ;; (latitude, longitude) coordinates.  E.g., look on
       ;; wikipedia, stackexchange, or even chatgpt.
       ;; The formula involves sin, cos, inverse cos, and other
       ;; arithmetic.
       ;; CHALLENGE: student must complete the implementation.
       (throw (ex-info "Missing one or more expressions, not yet implemented" {}))
       ;; HINT 6 line(s)
       ))
  )
```

Listing 9: *geodesic: geodesic-distance — the assignment explicitly permits looking up the formula.*

`cakecutting` Iteratively compute a sequence of shrinking pie-slice sizes, in two parallel styles (`loop`/`recur` and `reduce`), based on a puzzle whose closed-form solution students are explicitly told not to use — the point is the iterative computation, not the answer. The `loop`/`recur` variant:

```clojure
(defn piece-sizes-rec
  "This function, piece-sizes-rec, uses loop/recur to perform the
      calculation.

  returns a list of [k value] pairs where value is the size of
      the kth piece of pie.
  0.0 < value < 100.0 which indicates the percentage of the
      entire pie.  e.g. 0.5
  indicates half the pie.
  The list runs from n ... 1.
  The returned list is sorted into order of decreasing slice
      size."
  []
  (loop [n 1
         remaining-size 1.0
         acc ()]
    (let [piece-size (throw (ex-info "Missing single expression,
        not yet implemented" {}))
          ]
      (if (= n 100)
        (throw (ex-info "Missing single expression, not yet
            implemented" {}))
        (recur (throw (ex-info "Missing single expression, not
            yet implemented" {}))
               (throw (ex-info "Missing single expression, not
                   yet implemented" {}))
               (throw (ex-info "Missing single expression, not
                   yet implemented" {}))
               )))))
```

Listing 10: *cakecutting: piece-sizes-rec.*

`trig` Implement `exp`, `cos`, and `sin` from their Taylor series using `reduce`, including numerical-stability tricks (halving large arguments, reducing periodic arguments into range). `exp` is given fully worked out as the pattern to adapt for the other two — `cos` is the student's actual task:

```
(defn cos
  "
             1    x^2   x^4   x^6
   cos(x) = --- - --- + --- - --- + ...
             0!    2!    4!    6!
   Follow the pattern of 'defn exp' above to express the cosine computation
   in terms of reduce.  The difference is that you need to
    1) make sure the calculated term alternates in sign from one iteration
         to the next
    2) the range feeding into reduce is not from 1 to nTerms, but rather
         a different start and end point, and a step by 2
    3) given one term in the sum, what must you multiply it by
         to obtain the next?  It is no longer x/n as before.
    4) the iteration variable should traverse the even integers, (0, 2, 4 ...)

  This implementation contains an optimization.  If x < 0 then we compute cos(-x),
  and if x > 2pi, then we we compute cos(x - (2 n pi)) for some n which forces
  the argument of cos between 0 and 2pi.
  "
  [x]
  (let [pi Math/PI
        periods (Math/floor (/ x (* 2 pi)))]
    (cond (< x 0.0)
          (throw (ex-info "Missing single expression, not yet implemented" {}))

          (> periods 0)
          (cos (throw (ex-info "Missing single expression, not yet implemented" {}))
               )

          :else
          ;; we know 0 <= x <= 2pi
          (let [t0 1.0
                x2 (* x x)]
            (first
             ;; For clues to complete this excercise, look at the implementation of
                 `exp` above.
             (reduce (fn [[sum prev] n]
                       (throw (ex-info "Missing single expression, not yet
                           implemented" {}))
                       )
                     [t0 t0]
                     (range 2
                            (* 2 (dec n-terms))
                            2)))))))
```

Listing 11: *`trig`: `cos`, adapted from the fully-worked `exp`.*

`runaverage` A sliding-window running average over a list of samples, with a defined rule for the first few elements, before a full window is available:

```clojure
(defn run-average
  "run-average returns a value indicating the running average of a given
  list of floating-point numbers.  The average of how many numbers is determined
  by the given run-length.   E.g., if run-engtth==3, then we want to calculate
  successive averages of 3 consecutive numbers.
  run-average returns a sequence of pairs.
  The first component of each pair is the given number, i.e., the corresponding
      element of
  samples.  The second component of each pair is the average of 3 (or run-length)
  many elements of samples.
  There is an exception for the first run-length - 1 elements.  In this case
  you don't have run-length number of elements to average, so find the average
  of all the elements so far.
  Example, suppose samples = [1.0 2.0 6.0 4.0 2.4]
           and     run-length = 3
    this function should return a sequence equivalent to
    [[1.0  a1] [2.0  a2] [6.0  a3] [4.0 a4]  [2.4 a5]]
    where a1 = 1.0 / 1
          a2 = (1.0 + 2.0) / 2
          a3 = (1.0 + 2.0 + 6.0) / 3
          a4 = (2.0 + 6.0 + 4.0) / 3
          a5 = (6.0 + 4.0 + 2.4) / 3
  "
  [run-length samples]
  (loop [samples (seq samples)
         history () ;; history is ([sample average] ...)
         ]
    (if (empty? samples)
      (reverse history)
      ;; CHALLENGE: student must complete the implementation.
      (throw (ex-info "Missing one or more expressions, not yet implemented" {}))
      ;; HINT 8 line(s)
      )))
```

Listing 12: *runaverage: run-average.*

frenchnames Parses a real CSV of French baby-name statistics: replacing accented characters with their plain equivalents, counting occurrences of a name by year and gender, and finding hyphenated name variants. Mechanically simple once understood; the actual difficulty is careful handling of real-data edge cases, not a deep algorithm. The function with the single largest hint block of any assignment (24 lines):

```clojure
(defn baby-names-per-year
  ;; this function returns a Map which associates a year to count, where count is
  ;; the number of babies named name-target in that year of the specified
  ;; gender.  E.g., if
  ;; fred = (baby-names-per-year "fred" "M"), then (fred 2012)==100 means
  ;; there were 100 boy babies born named fred in 2012."
  ([name-target gender-target]
   (throw (ex-info "Missing single expression, not yet implemented" {}))
   )
  ([name-target gender-target elide]
   (assert (= name-target (lower-case name-target))
           (cl-format false "baby-names-per-year must be called with lower case name,
               not [~A]" name-target))
   (let [s (io/resource "France-baby-names/nat2020.csv")]
     (with-open [r (io/reader s)]
       ;; remember to drop the first line of file: sexe;preusuel;annais;nombre
       ;; skip lines where name is _PRENOMS_RARES
       ;; skip lines where the year is XXXX
       ;; names are in caps (e.g., FABIENNE) but nameTarget will be given as lower
           case (e.g., fabienne)

       ;; CHALLENGE: student must complete the implementation.
       (throw (ex-info "Missing one or more expressions, not yet implemented" {}))
       ;; HINT 24 line(s)
       ))))
```

Listing 13: *frenchnames: baby-names-per-year.*

`xlookup` A general CSV-to-CSV join, modeled on Excel's `XLOOKUP`: flexible column renaming, fan-out when multiple rows match, and a defined precedence rule when both tables share a column name. Familiar territory for anyone who knows SQL joins – the challenge is the volume of edge cases, not the underlying idea. The actual join logic:

```clojure
(defn join-csv
  "Return a pair of the form [[headers-1 headers-2]
                              seq-of-tables ]
  where each table corresponds to a line from csv-string-1 merged with a line
  from csv-string-2 where the value of field-1 of csv-1 equals the value field-2 of
      csv-2.
  If field-1 is different than field-2, then both appear in the output table.
  For example.
  if field-1 = \"x\"
  and field-2 = \"y\"
  from csv-1, line-1 = {\"x\" 100 \"b\" 200}
  from csv-2, line-2 = {\"y\" 100 \"c\" 300}
  --> {\"x\" 100 \"y\" 100 \"b\" 200 \"c\" 300}

  If a line from csv-1 does not correspond to any line in csv-2, then that line is
  omitted from the output seq.
  For example if we are goining on field-1 = \"x\" field-2 = \"y\", but
  and if a line line in csv-1 has \"x\" = 100, but no line in csv-2 has \"y\" = 100,
  then the line is omitted from the output.
  "
  [csv-1 parsers-1 rename-1 field-1
   csv-2 parsers-2 rename-2 field-2]
  (let [[headers-1 tbl-1] (read-csv-and-rename csv-1 :parsers parsers-1 :rename
      rename-1)
        [headers-2 tbl-2] (read-csv-and-rename csv-2 :parsers parsers-2 :rename
            rename-2)]
    [[headers-1 headers-2]
     (for [
           line-1 tbl-1
           :let [value-1 (get line-1 field-1)]
           line-2 tbl-2
           ;; CHALLENGE: student must complete the implementation.
           (throw (ex-info "Missing one or more expressions, not yet implemented" {}))
           ;; HINT 2 line(s)
           ]
       ;; compute a new map containing all the keys of line-1 and line-2.
       ;; if line-1 and line-2 have a common key, then the value from line-2
       ;; is used and the value from line-1 is discarded.
       ;; CHALLENGE: student must complete the implementation.
       (throw (ex-info "Missing one or more expressions, not yet implemented" {}))
       ;; HINT 1 line(s)
       )]))
```

Listing 14: *`xlookup`: `join-csv`.*

Three further templates are given but not graded, roughly in increasing order of difficulty: `calculus` (adaptive numerical integration and differentiation), `determinant` (a matrix determinant via Laplace expansion — the professor's own solution file for this one is marked incomplete), and `triptych` (the combinatorics of the card game SET, up through finding a maximal triptych-free "cap" — a genuinely hard combinatorial search, and by a wide margin the most difficult exercise in the entire set, deliberately left as an ungraded bonus challenge rather than a requirement).

3.2 Not All "Hard" Is the Same Hard

Two different kinds of difficulty show up across these assignments: figuring out *what* to do, and actually *building* it. They do not always travel together.

`reduce` is trivial to code, but the identity element for the pair-multiplication function it asks for is never stated anywhere; students have to reverse-engineer it from the test cases alone. Hard to understand, easy to code.

In lecture, this is where we walk through several ways of summing a sequence of numbers — plain recursion, `loop`/`recur`, `reduce` — and by the third technique, three hours into a session, even with a break, attention visibly drops. What reliably brings the room back is announcing that we are going to sum the sequence by partitioning it and adding each sublist in a separate thread. We write it using Clojure's `future`, and time it against the earlier approaches.

`factors` is the opposite. Everyone already knows the algorithm – prime factorization is taught in secondary school. The challenge is purely getting a correct trial-division loop written, and it is the single largest implementation effort of any required assignment. Easy to understand, hard to code.

Students may use any algorithm they like, but the intended one first factors out all the 2's: while `n` is even, divide by two and recur. Once `n` is odd, the remaining search only needs odd candidates, via `loop`/`recur` over four values: the accumulated primes, the current `n`, a lower bound, and an upper bound. Three optimizations turn this from merely correct into efficient. First, the upper bound is always round($\sqrt{n}$): no factor of a non-prime `n` can exceed its square root, so once the lower bound passes it, whatever remains of `n` is itself prime. Second, the search steps by 2, not 1 — every even candidate was already eliminated in the first step, so only odd candidates (5, 7, 9, . . .) need checking. Third, when the lower bound divides `n`, it may divide `n` more than once, so the next recursion keeps the *same* lower bound but divides `n` by it and recomputes a smaller upper bound; when the lower bound does *not* divide `n`, the next recursion moves to lower-bound + 2 with `n` and the upper bound left unchanged, since every smaller candidate has already been ruled out.

`geodesic` is hard on both axes. It needs the actual spherical-trigonometry formula for great-circle distance — the assignment itself tells students to look it up, "even ChatGPT" – and then several interlocking pieces (coordinate

conversion, latitude and longitude wraparound at the poles, a recursive great-circle bisection) all have to work together correctly.

The piecewise path itself is built by recursive bisection. Start with just the two endpoints and measure the distance between them; if they are already close enough, the sequence is done. Otherwise, insert a point roughly halfway between them and recur separately on the first half and the second half, then append the two resulting sequences. Each split roughly halves the distance between consecutive points, which is exactly the termination test, so the recursion is guaranteed to bottom out.

Finding that halfway point, though, has to happen on the sphere, not in latitude/longitude space. Each endpoint is converted to an `[x y z]` point in 3-D space; the two 3-D points are averaged coordinate-wise; and that average lands somewhere *inside* the sphere of the earth, not on its surface. Projecting it back out means treating it as a vector from the earth's center, normalizing it (scaling by the reciprocal of its own modulus), and rescaling the result by the earth's radius. Only then is the resulting `[x y z]` point — now genuinely on the geodesic — converted back to `[latitude longitude]`.

Traced on a planar projection, the shortest path between two points is a curve, not a straight line – Figure 1 shows why.

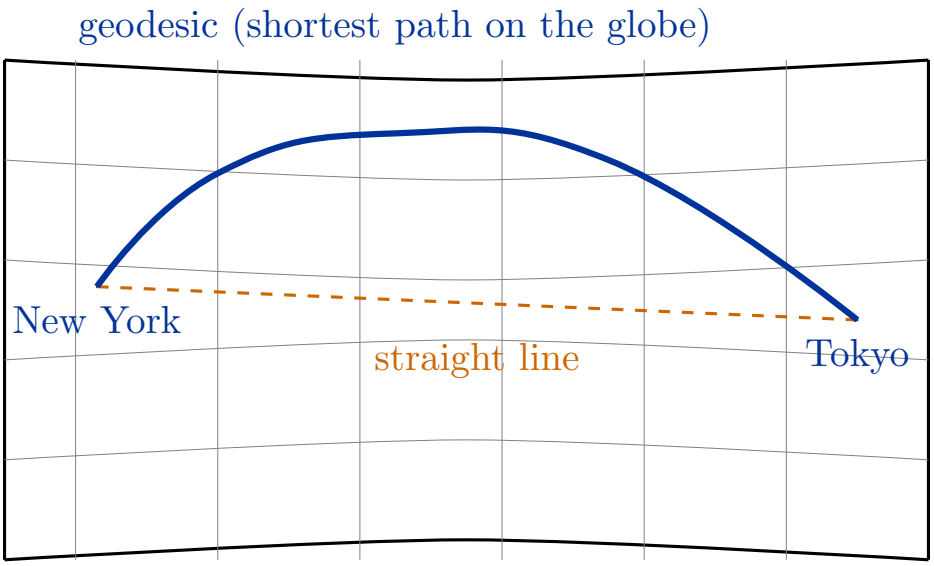


Figure 1: *A geodesic (the shortest path on a sphere) traced between New York and Tokyo, versus the straight line a planar projection suggests.*

Figure 2 is the real thing: actual geodesics, computed and rendered on a

real world map projection by a separate globe-plotting tool of mine, radiating from a point in southern Africa.

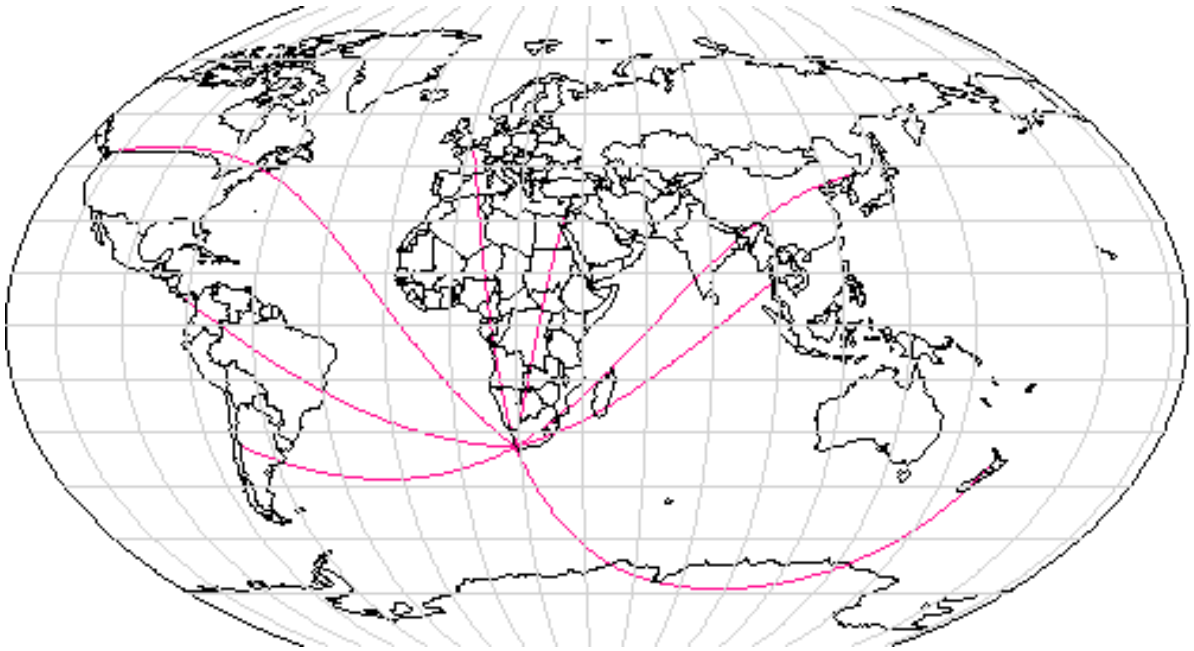


Figure 2: *Real geodesics rendered on an actual world map projection.*

Figure 3 places all three assignments on both axes at once: how hard the requirement is to understand, versus how hard it is to implement once understood.

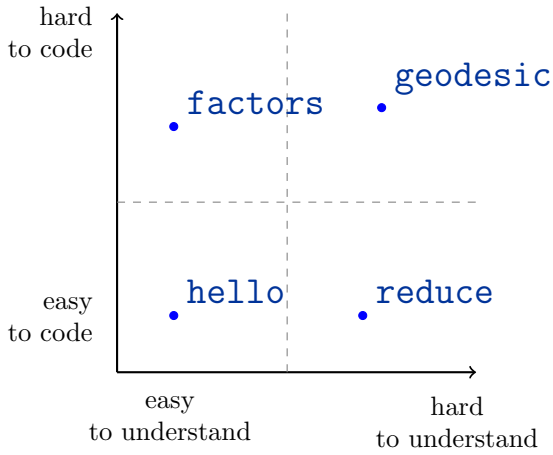


Figure 3: *`reduce`, `factors`, and `geodesic` placed by difficulty to understand versus difficulty to code — four of the course's thirteen required assignments.*

4 The Gap

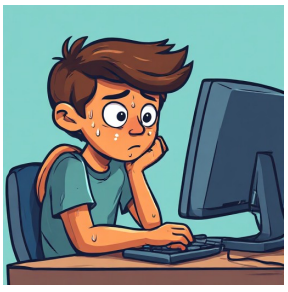
Most students do not fully "get it" by the end of the course. A few do. That difference is striking, and this section is an attempt to say something concrete about it rather than just assert it.

4.1 Method, and an Honest Caveat

Quiz 3 asked directly: *"What is the most difficult part of learning Clojure, given that you are already an expert programmer in other languages?"*, followed by a second question asking for a free comparison to any other language the student knew, and a consent question asking permission to use the student's comments and code in publications, with an explicit promise never to use the student's name.

Of the 13 students whose consent status is known and who agreed, only 4 left a substantive answer to the first question. This is not a survey with statistical power. It is a handful of real voices, and it should be read as exactly that — illustrative, not representative. Every quotation below is anonymized; no student is named or otherwise identified anywhere in this article.

4.2 What Fails to Land



All four answers gesture at "syntax" as the hardest part of learning Clojure — but they do not mean the same thing by it.

One student meant unfamiliar names: *"the unusual names of functions, like* `conj` *or* `contains?` *. . . checking for key and not value, which makes it harder to search through the docs, especially since the docs are incomplete for quite a few functions."* That second half is worth pulling out on its own: not everything here is Clojure being intrinsically hard for newcomers. Some of it is ordinary documentation debt, unrelated to the language itself, and exactly the kind of thing any project can chip away at.

Another meant the whole reading experience: *"clojure seams unlike the other languages... it's a functional language, just this makes it weird and hard... the whole syntax... makes it really hard to read and understand the code."* This is the deepest version of the complaint in the sample. It is not really about parentheses at all — it is a rejection of the functional model itself, not a readability complaint about how that model happens to be written down.

A third meant literal parentheses, plus something else entirely: *"syntax (including parenthesis)... we will have to rely on another language like Java"* (Clojure's Java interop, needed for the "functional core" architecture the course discusses). Three students, three different obstacles, one shared word.

4.3 What Clicks

The fourth student's entire answer to "what was the hardest part" was: *"Nothing."*

This student's comparison paragraph explains lazy sequence evaluation, and frames manual versus automatic memory management as *both* a strength and a weakness of Clojure's hosted, JVM-garbage-collected model — a genuinely sophisticated observation. Even the one complaint about parentheses reads like a joke from someone comfortable, not a frustration from someone stuck: *"(((Too) Many) Parentheses)"*.

It would be a mistake to read the other three responses as purely negative, though. Understanding and struggling coexist more often than a clean fails-to-land/clicks split suggests. The student who found naming and documentation hardest still volunteered real appreciation for the JVM ecosystem and specifically for `#(...)` lambda syntax. The student who found Java interop hardest still produced a careful, accurate, unprompted four-point technical comparison to Python (parenthesization, data-literal syntax, action-first ordering, immutability). Most students are somewhere in between the two poles. The clean cases — entirely stuck, or entirely fluent — are the exception, which is exactly why they stand out enough to discuss here.

Same course, same four sessions, same assignments. What actually makes

the difference is the subject of the rest of this article.

5 Curriculum Constraints

Seven of the fourteen students who submitted anything never submitted a single quiz — meaning, as far as attendance can be measured, they never once came to a class session. Of those seven, six scored between 92% and 100% on the graded homework.

The student quoted in Section 4 who answered "Nothing" came to every single session, and wrote the most fluent, technically sophisticated response in the sample. That same student has the *lowest* homework score in the class. Looking closer, the shape of that low score is itself informative: this student scored 100% on exactly the first six required assignments attempted, and simply never submitted the remaining seven. Not a struggle scattered across many assignments — a clean stop, partway through, apparently by choice.

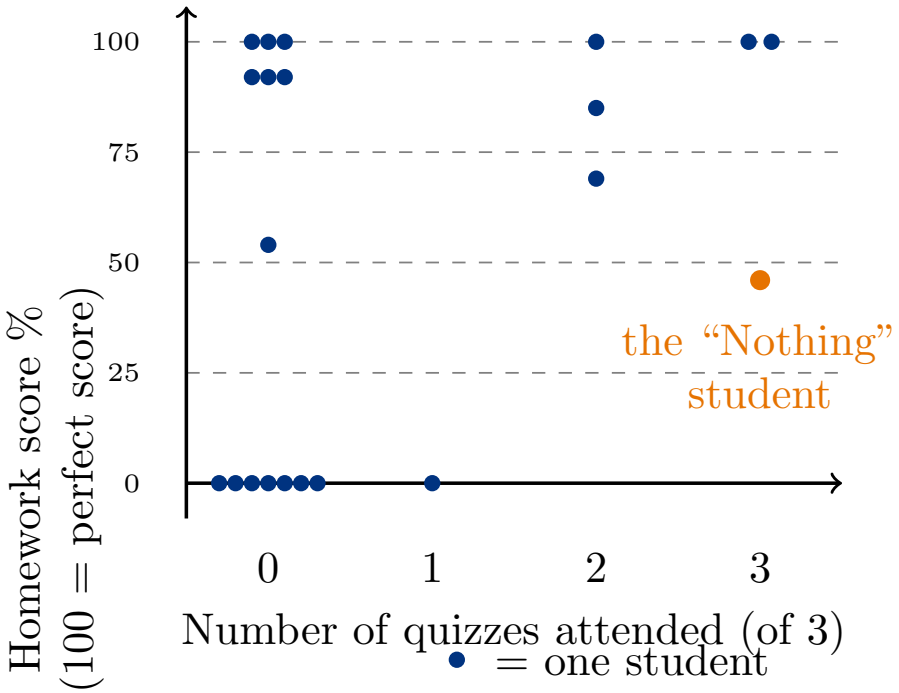


Figure 4: *Homework score versus quizzes attended, 2026. The "Nothing" student is the one point at full attendance and a mid-range score.*

5.1 More Than One Explanation

A low homework score does not have to mean a student struggled. Students are told explicitly that they do not need 100%, that quizzes are intentionally easy, and this is an elective running concurrently with other courses' large, due-at-semester-end projects. Doing every homework problem, especially the harder ones, costs time some students genuinely need elsewhere. A low score can simply mean: "I did enough to validate the course, and spent my remaining hours on something else."

It can also mean an LLM did the work with minimal understanding behind it. From the data alone, we cannot tell these two explanations apart – and we do not think a per-submission guess based on code style would be reliable enough to trust either, a point Section 6 returns to directly.

6 Academic Integrity: Can We Tell?

Two original fears motivated this section, going into the course. First, that an LLM would do a student's homework, with little understanding behind it. Second, that because the assignments repeat, similar but not identical, year to year, a student might simply find a prior year's solution published somewhere and copy it.

No mechanism exists to confirm or rule out either one for any *individual* submission. That is not a caveat to slide past — it is the honest starting point for this section. What follows is what we could establish about the class as a whole, which is a different and much weaker kind of claim than being able to say anything about one student's one submission.

6.1 Checking the Easier Fear First

For the second fear, at least, a real check is possible: search the way a student actually would. We searched for the course and repository names, and for distinctive function names pulled directly from the assignment templates (`bin-search-by-boolean`, `piece-sizes-rec`, `piece-sizes-reduce`, `make-adj-directed`, `reachable-vertices`), plus the homework filenames themselves (`frenchnames.clj`, `geodesic.clj`, `triptych.clj`). Nothing

came back: no blog posts, no public forks or copies, no indexed repository under that name containing this content.

A clean web search only means nothing *indexed and crawlable* turned up. It says nothing about a private Discord server, a closed Gist, or a group chat — exactly where student-to-student sharing would actually happen if it happens at all. We checked the obvious public route, and it is clean. That is all this rules out.

6.2 Every Metric, Same Direction

For the first fear, we do not have a way to check any individual submission, but we do have three years of the same course, on the same MAAS grading pipeline (Section 2.3), which makes an aggregate comparison possible.

Where these numbers come from

Every tag a student ever pushes is a permanent record on the grading platform: which assignment, which student, when, whether the grader finished running, and the score if it did. The platform's own command-line tool can export that complete history for an entire course in a single call, as one row per pushed tag. That export is private to course staff — nothing in this section is something a reader could go reproduce themselves, since it depends on an account on a grading platform this article's audience does not have access to — but the method itself is ordinary and does not depend on anything EPITA-specific: any course built on a similar auto-grading service (GitHub Classroom, an internal CI-based grader, *etc.*) records the same underlying facts, and the same procedure would apply.

From that export, for each of the three years, we built the table below in four steps. First, keep only the assignments that were actually required that year — not the final exam, and not our own testing submissions, which the platform logs exactly like a student's push and does not otherwise distinguish. Second, keep only pushes the grader finished evaluating, whether it passed or crashed; a push still mid-evaluation at the moment of export is neither and is dropped. Third, group the remaining pushes by (student, assignment), ordered by when each was received. We call one such group a **student-assignment**: one specific student's whole attempt history on one specific assignment, however many pushes that took — this is the unit the rest of this section counts and computes over. Fourth, compute the four statistics

below over those groups.

One wrinkle worth naming because it is exactly the kind of thing that silently corrupts a count like this: the tag name a student chooses is not a reliable identifier. Two pushes with the same tag name are not necessarily the same attempt re-recorded — the same name can legitimately be reused by the same student for two genuinely separate pushes, sometimes a day apart. Counting by tag name instead of by the platform's own internal row identifier silently collapses real retries into one, which is precisely the kind of small methodological error that would understate exactly the effect this section is trying to measure. For every required assignment, for every student across 2024, 2025, and 2026, we counted how many such genuine attempts were pushed before the assignment passed, and cross-referenced against each year's own recorded grade for that assignment.

Year	1-try rate	avg. tries	1-try → 100%	eventually 100%
2024	63.1%	2.39	83.6%	65.6%
2025	65.2%	1.86	92.3%	68.0%
2026	76.6%	1.47	**100.0%**	91.9%

Table 1: *Attempt counts and outcomes per required assignment, by year. "1-try rate" is the fraction of student-assignments settled in a single pushed attempt. "1-try → 100%" is, among those single-attempt student-assignments, the fraction that scored perfectly. "Eventually 100%" is the same statistic computed over the multi-attempt student-assignments.*

Reading the table and chart

Every row is one year of the course; every column is a statistic computed over all student-assignments that year, pooled across every required assignment and every student. A student-assignment's *attempt count* is simply how many git tags (submission attempts) the student pushed for that assignment before they stopped pushing – whether or not that final push actually passed. A single-attempt student-assignment is therefore *not* necessarily a success: in 2024, for example, one student pushed `CakeCutting` once, scored 33%, and never pushed it again. Three things worth being explicit about, since they are easy to misread at a glance: every statistic here is per *assignment*, not per student's overall course grade – a student contributes one data point per required assignment, not one data point for the whole course. "1-try

rate" counts every single-push student-assignment regardless of whether that push succeeded. And "eventually 100%" is *not* computed over all student-assignments; it is computed only over the student-assignments that needed more than one attempt, i.e. the complement of the "1-try" group.

Column by column:

1-try rate Of all student-assignments that year, the percentage that were settled in exactly one pushed attempt – the student pushed once and did not need, or did not choose, to retry.

avg. tries The mean number of attempts per student-assignment, across every student-assignment regardless of outcome.

1-try → 100% Narrowed to only the 1-try student-assignments from the first column: of those, the percentage that scored a perfect 100% on that single attempt. This answers "when a student only tried once, how often was that one try already correct?"

eventually 100% Narrowed instead to the student-assignments that needed *more than one* attempt: of those, the percentage whose final, last-pushed attempt eventually scored 100%. This answers "when a student needed to retry, how often did they eventually land on a perfect score?"

The unlimited-resubmission policy matters here. MAAS enforces no cap on attempts, so the cost of pushing an incomplete draft is zero. The natural workflow that invites — submit something rough, read which tests fail, fix, resubmit — does not require reading the assignment spec carefully before the first push; it substitutes iteration against test failures for up-front comprehension. That is not a criticism of students: it is how many programmers, including us, actually approach an unfamiliar problem. Under that baseline, a low 1-try rate and a low 1-try → 100% rate are what we would expect by default. A submission that is both single-try and perfect requires having understood the assignment correctly *before* seeing any test failure — which is precisely what the natural workflow bypasses. That is what makes the high, and rising, 1-try → 100% column harder to wave away as organic behavior.

Figure 5 plots the three percentage columns as lines across the three years, which makes the shared direction easier to see at a glance than reading four numbers per row: all three lines rise, every year, with no reversals.

Every number in Table 1 moves the same direction, every year, with no reversals. The most striking cell is the bottom right of the "1-try" column:

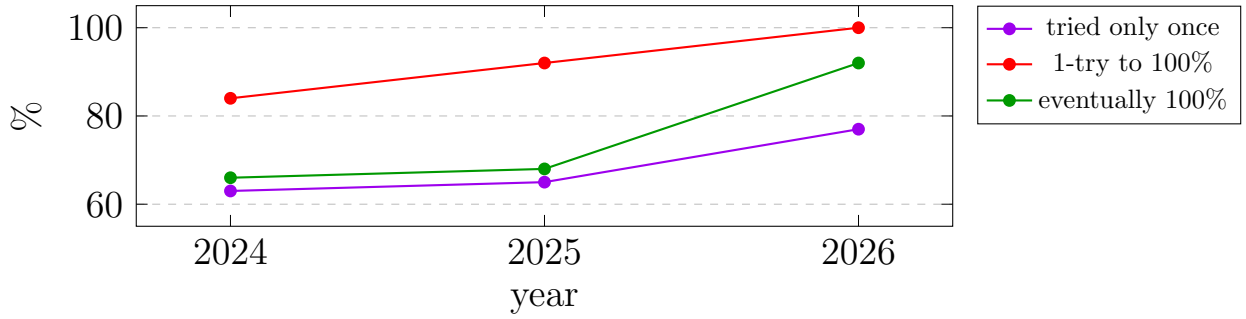


Figure 5: *The three percentage columns of Table 1, plotted by year. All three move the same direction, every year.*

in 2026, *every single* student who only pushed an assignment once got it right. Zero exceptions. In 2024 and 2025, pushing once and still being wrong happened regularly — 16% and 8% of the time respectively. Even the multi-attempt submissions increasingly resolve to a correct answer (92% this year versus roughly two-thirds before) rather than plateauing on something broken.

This is exactly the shape you would expect if LLM assistance became more common and more capable, year over year: fewer failed attempts, and a first attempt now essentially guaranteed correct. It is still only correlational. Better documentation, clearer assignments, or a stronger cohort are none of them ruled out by this data, and it says nothing at all about any individual student — only about the class as a whole, over time.

A smaller, more granular version of the same pattern: across all of 2024 and 2025 combined (62 students), not one shows a submission history consisting of a single commit to the whole repository with every required assignment tagged as a first attempt and no retries anywhere. In 2026, three of fourteen students do. That specific pattern has no precedent in two years of prior data and appears in about a fifth of this year's cohort.

6.3 Does This Generalize?

Everything until now has been about one course. Before treating it as anything more than an anecdote about Clojure specifically, the obvious next question is whether the same shape shows up anywhere else we teach.

We picked the other courses on the same grading platform where we have

at least two years of history: two Scala offerings of the same functional-programming elective this one is descended from (an engineering-track and an apprenticeship-track version, taught in parallel to different student populations), a graph theory course, and three courses that are not really programming courses at all — an introductory algebra course, a linear algebra course, and a data structures course, all taught in Python to a different student body (an international Bachelor's program) than the Clojure elective. Programming there is the mechanism for checking whether a student understood the actual subject — graph algorithms, vector spaces, row-reduction — not the subject itself. If the earlier trend is really about something as general as "LLMs got better and more available," it should not care whether the assignment is a Clojure homework or a linear algebra exercise checked by running Python code. If it only shows up in Clojure, that is itself informative.

The two Scala offerings are worth a word up front, since they appear below as two separate rows rather than one: same course, same assignments for the most part, run in parallel to two entirely separate, non-overlapping student populations — though the apprenticeship track drops several of the harder assignments. We kept them as two rows rather than pooling them, because pooling turns out to make the trend *less* legible, not more: the two tracks were not both running in every year, so a merged series would mostly reflect which population happened to be enrolled that particular year, not a real year-over-year change.

Two courses we also have Forge history for turned out not to be usable here and are omitted: one has essentially a single semester-long assignment rather than a series of discrete ones, which makes a per-assignment "1-try rate" meaningless; the other's most recent year had no real student submissions yet at the time of writing — every row was our own testing account, not a student's.

The Shape of the Distribution, Not Just the Average

Every statistic so far is an average or a percentage computed over student-assignments. It is worth looking at the raw shape of how many tries a student-assignment actually took, across the full pooled set of 9,486 student-assignments behind Figure 7: the mean is 2.15 tries — unsurprising, since that is the same ratio "avg. tries" already reports elsewhere. The median, though, is exactly *1*.

More than three in five student-assignments — 5,756 of 9,486, 60.7% —

Course	Year	student-assignments	1-try	avg. tries	1-try→100%	eventually 100%
Clojure	2024	339	63.1%	2.39	83.6%	65.6%
	2025	279	65.2%	1.86	92.3%	68.0%
	2026	158	76.6%	1.47	100.0%	91.9%
Scala (eng.)	2023	555	77.1%	1.48	88.3%	72.4%
	2025	254	70.9%	1.65	92.2%	81.1%
Scala (appr.)	2024	712	66.4%	1.76	97.5%	91.6%
	2025	1030	68.4%	1.98	98.0%	90.5%
	2026	951	77.7%	1.52	97.7%	89.2%
Graph theory	2024	569	39.5%	3.15	73.8%	71.2%
	2025	433	45.5%	2.91	84.8%	74.6%
Algebra	2023	337	51.6%	2.59	57.5%	62.0%
	2024	406	36.9%	3.02	74.7%	74.2%
	2025	714	43.0%	2.90	89.3%	86.7%
Linear algebra	2024	483	53.8%	2.17	80.4%	67.3%
	2025	583	59.7%	1.97	90.2%	86.8%
	2026	806	66.4%	1.61	94.8%	90.8%
Data structures	2025	406	54.2%	3.05	95.9%	88.7%
	2026	400	63.7%	1.96	88.2%	87.6%

Table 2: *The same four statistics as Table 1, computed the same way, for six more courses. "Scala (eng.)" and "Scala (appr.)" are the same elective taught in parallel to two different student populations.*

Half-year	student-assignments	tried once	1-try→100%	eventually 100%	share of subm.
Fall 2023	337	51.6%	57.5%	62.0%	4.3%
Spring 2024	2089	65.8%	89.2%	76.0%	19.4%
Fall 2024	975	38.4%	74.2%	72.5%	14.9%
Spring 2025	2146	65.9%	94.6%	85.4%	20.3%
Fall 2025	2504	58.4%	93.9%	84.8%	29.7%
Spring 2026	1364	66.8%	93.6%	89.9%	11.4%

Table 3: *DRAFT: the exact values behind Figure 7. Each row pools every course active that half-year (Table 1's Clojure rows and Table 2's six other courses, placed onto a half-year per the source comment above), weighting every submission equally rather than every course equally. "Student-assignments" and "share of subm." sum to 9,415 and 100% respectively across all six rows.*

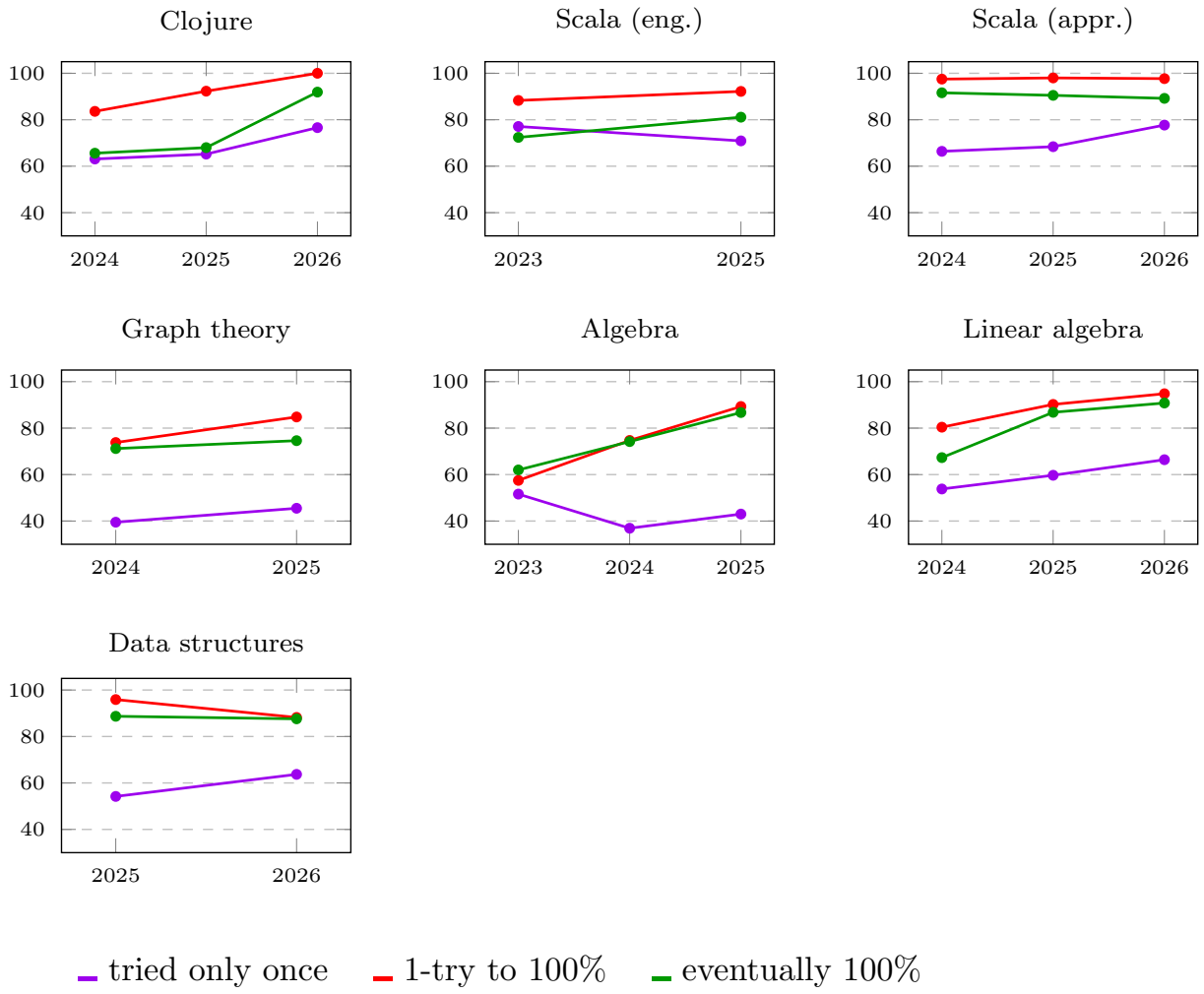


Figure 6: *The same three percentage columns from Table 2 (and, top left, Clojure again from Table 1, for reference), plotted the same way as Figure 5, one panel per course. Note the panels do not share an x-axis — each shows that course's own years, which do not all start or end together.*

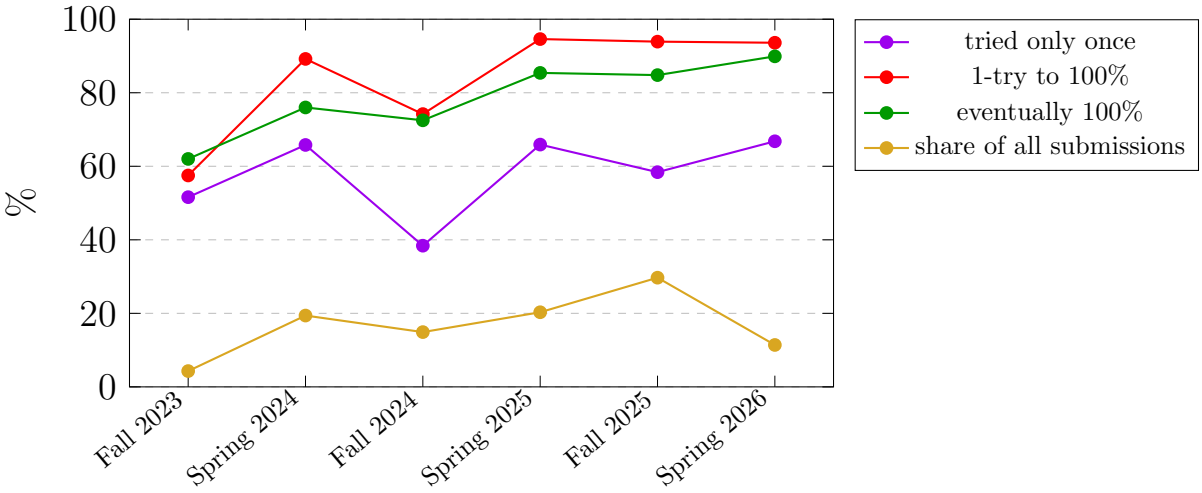


Figure 7: *DRAFT: exact values in Table 3. The same three percentage columns as Figure 6, pooled across every course active in a given calendar half-year instead of kept per-course (every submission weighted equally, so larger courses count for more), and plotted against actual calendar time instead of each course's own year-index, plus a fourth curve showing what share of all submissions fell in that half-year (so a dip or spike in the other three curves can be checked against how much data actually backs it). Computed from 9,415 student-assignments total (Tables 1 and 2); a live re-audit of the same eighteen course-years puts the exact total at* 20,427 individual submissions *(a few of those courses are still active and picked up a handful more real pushes since the tables above were computed). See source comment for how each course-year's semester was determined.*

were settled in a single push. Table 4 shows how sharply it falls off from there.

Tries	Student-assignments	% of student-assignments
1	5,756	60.7%
2	1,592	16.8%
3	811	8.5%
4	439	4.6%
5	266	2.8%
6	173	1.8%
7	117	1.2%
8	76	0.8%
9	55	0.6%
10	53	0.6%
11+	148	1.6%

Table 4: *Number of tries per student-assignment, pooled across all 9,486 student-assignments behind Figure 7. Mean: 2.15. Median: 1.*

That median is the genuinely surprising number, given the incentive structure described earlier: unlimited, free resubmission should reward a submit-something-rough-then-iterate workflow, which does not require reading the assignment spec carefully before the first push. If that workflow were the norm, the *typical* student-assignment should need several tries, not one — the median should sit comfortably above 1. Instead, the single most common outcome, by a wide margin, is getting it right on the very first attempt. That is not the shape of a genuinely exploratory, trial-and-error process; it looks much more like what you would expect if a large fraction of students already had a correct or near-correct solution in hand before they ever saw a test failure.

Which ones actually match

Three of the six — both Scala tracks and graph theory — move in the same direction as Clojure on most or all four columns, year over year. Linear algebra matches cleanly on all four, every year. The other three are messier: algebra's 1-try rate and average-tries columns are not monotonic across its three years, even though its two success-rate columns rise sharply (57.5% → 89.3% and 62.0% → 86.7%); the apprenticeship-track Scala course's eventually-100% column actually *falls* slightly across its three years (91.6% → 89.2%) while

its 1-try rate still rises; data structures only has two years to compare and is mixed on those. This is not the clean, every-column, every-year confirmation Clojure showed. Three courses reproduce it, three do not, and none of them contradicts it outright either.

A confound worth naming: it's the same students

Algebra, linear algebra, and data structures are not independent data points. Checking course rosters directly: the students who take algebra in a given year are — with well over 90% overlap in every cohort we checked — the same students who take linear algebra and data structures together the following year. Comparing algebra's year-over-year trend against linear algebra's is not really comparing two different populations; it is comparing the same population's first year against their second.

That makes a direct, individual-level check possible in a way nothing else in this article does: for every student who appears in both a given algebra cohort and the following year's linear algebra course, we can compare that one person's own 1-try rate across the two courses, not just the class average. Doing this for the two cohorts where both years exist: 32 of 36 students (89%) and 41 of 48 students (85%) individually improved their own 1-try rate from algebra to linear algebra the next year; none stayed exactly the same. The class-level average moved from 37.8% to 59.8% for one cohort and 45.4% to 68.0% for the other. This is not a cohort-composition artifact — it is most of the same individuals genuinely needing fewer attempts a year later. Whether that is growing comfort with the tools, the material itself, increasing LLM reliance, or some mix, this data alone cannot say — but it does rule out "different students, different course" as the explanation for algebra's messier trend.

Even the easiest assignment is noisy

One more honesty check, prompted by how bad algebra's own `Hello` assignment — the trivial warm-up every course in this study opens with, worth essentially nothing on its own — looked in isolation: 19% and 4% one-try rates in its two most recent years, the worst of any assignment in either course, every year. Trivial content, terrible numbers. That is not a difficulty signal; a warm-up assignment used to confirm the submission pipeline works has no real content to be difficult. It is friction from the tooling itself — a forgotten commit, an unsaved file, a student who cloned their workarea

twice without realizing it and pushed from the wrong copy. Excluding `Hello` entirely raises every course's numbers by a few points, but it does not close the gap between algebra and linear algebra, so it is not the explanation for that — it is a reminder that "how many tries did this take" is measuring some amount of pure administrative noise alongside whatever it says about difficulty or LLM use, and that noise is largest exactly where you would least expect it: the assignment with nothing to get wrong.

6.4 What We Still Don't Know

We can see that something changed, in Clojure and in some but not all of the other courses checked the same way. We still cannot tell you what, for any given piece of code in front of us.

7 Conclusion

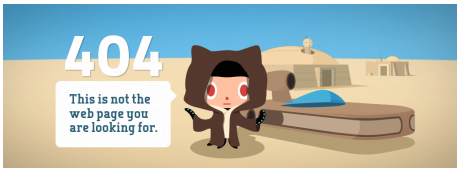
The abstract for the talk this article accompanies was written months before the course ran, and it promised a story about immutability, recursion, and lazy evaluation. What actually turned out to matter more was more mundane: attendance economics, students quietly deciding how much effort a given grade is worth, and a three-year pattern in submission data we still cannot fully explain. We think that is a more honest article than the one we originally set out to write.

- **Partial success:** some students clearly get it – completely, and quickly. Live coding, with real errors made and debugged in front of the room, seems to be part of why.
- **Real insight:** the divide between attendance and homework score is real and measurable. The three-year trend in how submissions get made is real too, and new to this year specifically.
- **Open questions:** we still cannot tell, for any single submission, why a score is low — time triage, an LLM, or something else entirely. We do not know how to solve that.

That last uncertainty is narrow – about one submission at a time. A broader set of open questions sits above it, and we do not have answers to

any of them: What should we, as instructors, do about student use of LLMs? Are hand-written programming exercises like these still useful, or already out of date? What does a student's grade even mean, under these conditions? Is learning to program itself becoming obsolete? We would like to hear other perspectives on any of these.

There is also a more mundane risk worth naming: the course's bus factor is currently one. $\Phi_{bus} = 1$ — the support material, beyond what is written down in the assignment templates themselves, is mainly in my head. What documentation exists is inconsistent and incomplete. I would like to publish the course and its exercises in a more reusable, self-contained form; whether an LLM can help with that serialization is itself an open question.



The course repository itself is private, deliberately, so students cannot simply find the solutions on GitHub. If you are a Clojurian and would like to take a look, I am happy to grant access – email me at jimka.issy@gmail.com, find me as Jim Newton on the Clojurians Slack, or as jimka2001 on Discord and GitHub.

If you were teaching this course, what would you want to know that I don't? I would like to hear it.