Procedural Generation of Nonverbal Reasoning Items: A Functional-Programming Architecture for IQ-Style Test Construction

Jean-François Parent
jfparent.dev@pm.me
https://www.supersharp.quest/super-sharp
September 2022

Abstract

Nonverbal reasoning items—progressive matrices, symbol-mapping tasks, and related figural puzzles—are a mainstay of intelligence testing, yet they are expensive to author by hand and, once published, leak into the public domain and into machine-learning training sets, eroding their measurement value. We present the generator subsystem of *super-sharp*, a ClojureScript application that procedurally synthesises such items. The design rests on three ideas. First, a *pattern generator* expresses each item's rule as a pure closure over lazy, infinite sequences: a total function from ordinal cell position to visual content, so a few lines of code yield an unbounded supply of distinct items. Second, an *answer-perturbation generator* produces distractors by three composable strategies—over-sampling the generative rule, structural add/remove mutation, and symbol substitution—each of which keeps the correct answer present and unique. Third, a *logical-validation* layer built on the Malli schema library constructs an effective schema from each item's own keys and enforces structural solution-integrity before rendering. We describe the architecture, give three deep case studies, tabulate nine generator families, and close with the limitation that psychometric calibration remains future work.

Keywords: procedural content generation, item generation, Raven's Progressive Matrices, functional programming, lazy sequences, schema validation, Clojure.

1. Introduction

Figural reasoning tests such as Raven's Progressive Matrices (RPM) [1, 2] ask a respondent to infer a generative rule from a partially filled matrix and select the cell that completes it. Their appeal as a "culture-reduced" measure of fluid reasoning is well documented [3], but two practical problems limit their use. First, authoring items by hand is slow and requires expert judgement to control difficulty. Second, a fixed bank of items is a depleting asset: once items circulate on the web—or enter the training corpus of a language model— re-use compromises validity.

Automatic item generation addresses both problems by treating an item not as an artefact but as a *sample from a generative process* [4, 5, 6]. If the process is correct by construction, an arbitrary number of fresh, well-formed items can be drawn on demand.

This paper documents the generator subsystem of *super-sharp*, an application that renders such items to screen and to print-ready PDF. Our contribution is not a new psychometric theory but an *architecture*: a compact, functional design in which

- **rules are functions, not data** — each item family is a closure over lazy sequences, sampled at successive positions (Section 4–5);
- **distractors are derived, not hand-picked** — three perturbation strategies generate wrong-but-plausible options while preserving a unique correct answer (Section 6);
- **validity is enforced mechanically** — a schema derived from each item's own structure rejects malformed output before it reaches the renderer (Section 7).

Section 8 catalogues the nine generator families in production; Section 9 states limitations, chiefly that *validation* here means logical solution-integrity, not empirical difficulty calibration.

2. Background and related work

Raven's Progressive Matrices. Carpenter, Just and Shell [3] gave the canonical cognitive analysis of the Advanced Progressive Matrices, decomposing items into a small set of relational rules (constant in a row, quantitative pairwise progression, figure addition/subtraction, distribution of values). Their rule taxonomy is the conceptual template for the pattern generators in Section 5: our "progression" and "addition" families correspond directly to their categories.

Automatic generation. Embretson's cognitive design-system approach [6] showed that item difficulty can be manipulated by controlling the number and type of rules, making generation a route to *validity by design* rather than post-hoc calibration. Matzen et al. [5] built "Sandia" matrices by systematically varying rule parameters. Most relevant to us, Wang and Su [4] formalise RPMs in first-order logic and *restrict instantiation to valid problems*, generating hundreds of well-formed items per second and showing, in a user study, that synthetic items were statistically indistinguishable from authored ones. Their central lesson — enforce well-formedness at generation time — motivates our validation layer (Section 7), though we enforce it with data-schema decoding rather than a logic solver.

Procedural content generation and functional programming. Viewed from games research, item synthesis is constructive procedural content generation with a correctness constraint [7]. Our implementation leans on Clojure's lazy sequences and first-class functions [9]: an infinite `cycle` of shapes or colours is a value that costs nothing until sampled, and a generative rule is an ordinary closure. Validation uses Malli [8], a data-driven schema library for Clojure/Script that lets a schema be *computed* at run time from the item's shape.

3. System architecture

Every generator family exposes the same three-function skeleton—`gen`, `gen-parts`, and `gen-pattern`—and produces a single, renderer-agnostic data structure. Figure 1 shows the pipeline.

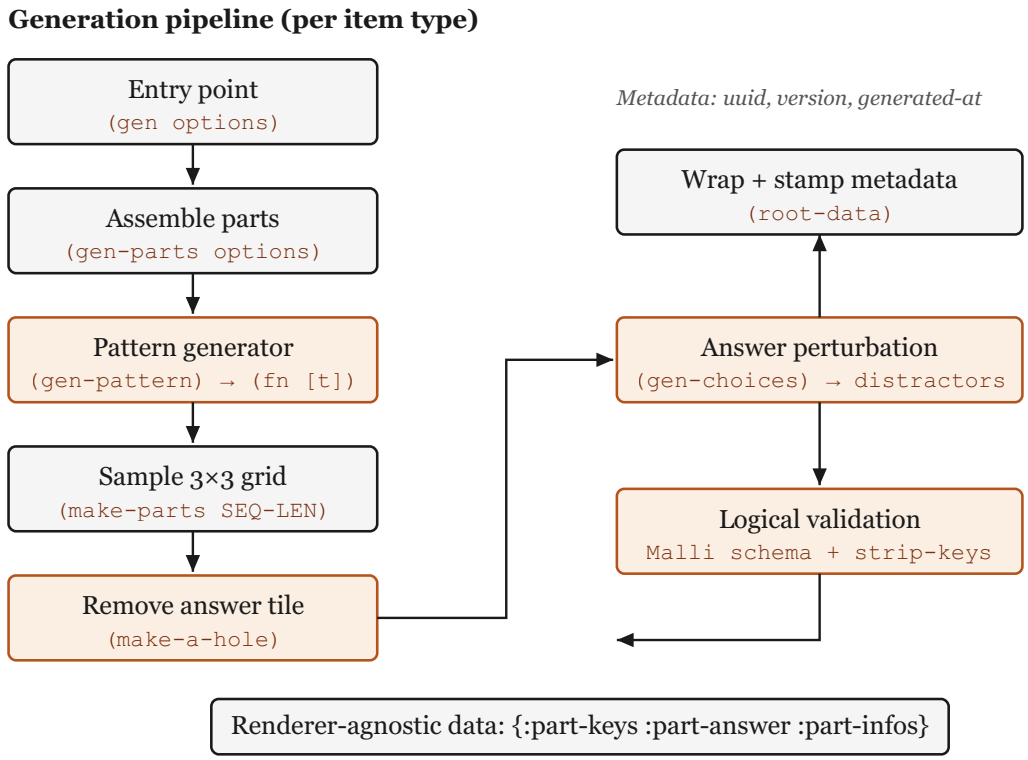


Figure 1. The shared generation pipeline. `gen-pattern` builds a rule closure; `make-parts` samples it into a 3×3 grid; `make-a-hole` extracts the answer; `gen-choices` derives distractors; `root-data` stamps metadata; and a Malli pass normalises the result into keys the renderer understands.

The top-level `gen` for the colours-and-shapes matrix is representative (`generators/progressive_matrix_colors_and_shapes.cljc`):

```
(defn gen-parts [options]
  (let [pattern-fn (gen-pattern options)
        parts      (u/make-parts SEQ-LEN pattern-fn "q")
        [holed-parts hole] (u/make-a-hole parts)
        choices    (u/gen-choices pattern-fn hole CHOICES-LEN SEQ-MAX)
        sorted-keys (into (u/sort-keysmap-v holed-parts)
                          (u/sort-keysmap-v choices))
        part-infos  (merge holed-parts choices)]
    [sorted-keys part-infos hole]))

(defn gen [options]
  (let [[part-keys part-infos answer] (gen-parts options)]
    (-> (ssdata/root-data
         {:id :progressive-matrix-colors-and-shapes
          :version 1
          :name "Progressive Matrix - Colors & Shapes"
          :part-answer answer
          :part-keys part-keys
          :part-infos part-infos})
        ssdata/cleanup-progressive-matrix-config)))
```

Listing 1 — The shared skeleton: build a rule, sample a grid, punch a hole, derive choices, wrap, validate.

`root-data` (in `libs/ssdata.cljs`) wraps the payload with a UUID, a schema `:version`, and a `:generated-at` timestamp, then exposes three fields the renderer consumes: `:part-keys` (ordered tile identifiers `:q0…:q8`, `:c0…:c5`), `:part-answer` (the correct tile's configuration), and `:part-`

`infos` (a map from key to a vector of drawable shape maps). Because the output is plain data, the generator is fully decoupled from the fabric.js canvas renderer and the PDF exporter; the same item can be drawn in either.

4. The pattern generator

The heart of the design is that a *rule is a function of position*. `make-parts` (in `utils.cljc`) turns any such function into a grid of tiles:

```
(defn make-parts
  "Generate config from pattern.
   n => number of parts; pattern-fn => (fn [t]); key-prefix => \"q\" or \"c\"."
  [n pattern-fn key-prefix]
  (apply hash-map (mapcat
                   #(into [(keyword (str key-prefix %))
                           (pattern-fn %)])
                   (range n))))
```

Listing 2 — `make-parts` samples a pattern closure at positions 0…n−1 (`utils.cljc`).

The closure itself is produced by `gen-pattern`. For the colours-and-shapes family it draws shapes and colours from *lazy infinite sequences* and indexes them by the cell position `t`:

```
(defn gen-pattern [options]
  (let [get-color-seq (fn [colors]
                        (->> colors (take (u/srand-int 1 5)) shuffle cycle))
        color-packs   (cycle (sscolor/get-color-packs))
        obj1-shapes   (take SEQ-MAX (ssshape/get-shapes-generator options))
        obj1-inner    (take SEQ-MAX (cycle (get-color-seq (first color-packs))))
        obj1-border   (take SEQ-MAX (cycle (shuffle (nth color-packs 1))))
        obj2-shapes   (take SEQ-MAX (ssshape/get-shapes-generator options))
        obj2-inner    (take SEQ-MAX (cycle (get-color-seq (second color-packs))))]
    (fn [t]
      [{:shape (nth obj1-shapes t) :size [0.45 0.45]
        :inner-color (nth obj1-inner t) :border-color (nth obj1-border t)}
       {:shape (nth obj2-shapes t) :size [0.1 0.1]
        :border-color "black" :border-size 2
        :inner-color (nth obj2-inner t)}])))
```

Listing 3 — A rule closure over lazy sequences (`progressive_matrix_colors_and_shapes.cljc`).

Two ideas do the work. `ssshape/get-shapes-generator` returns `(cycle (take k (shuffle SHAPES)))` — an endless repetition of a random k-shape motif — and `cycle` over shuffled colour packs does the same for colour. Sampling `(nth seq t)` at `t = 0…8` (row-major) yields a matrix in which each visual channel advances on its own period; the respondent must infer those periods. Because the sequences are infinite and lazy, the code never enumerates more than it samples, and a single definition produces a combinatorially large item space. Figure 2 illustrates the sampling model.

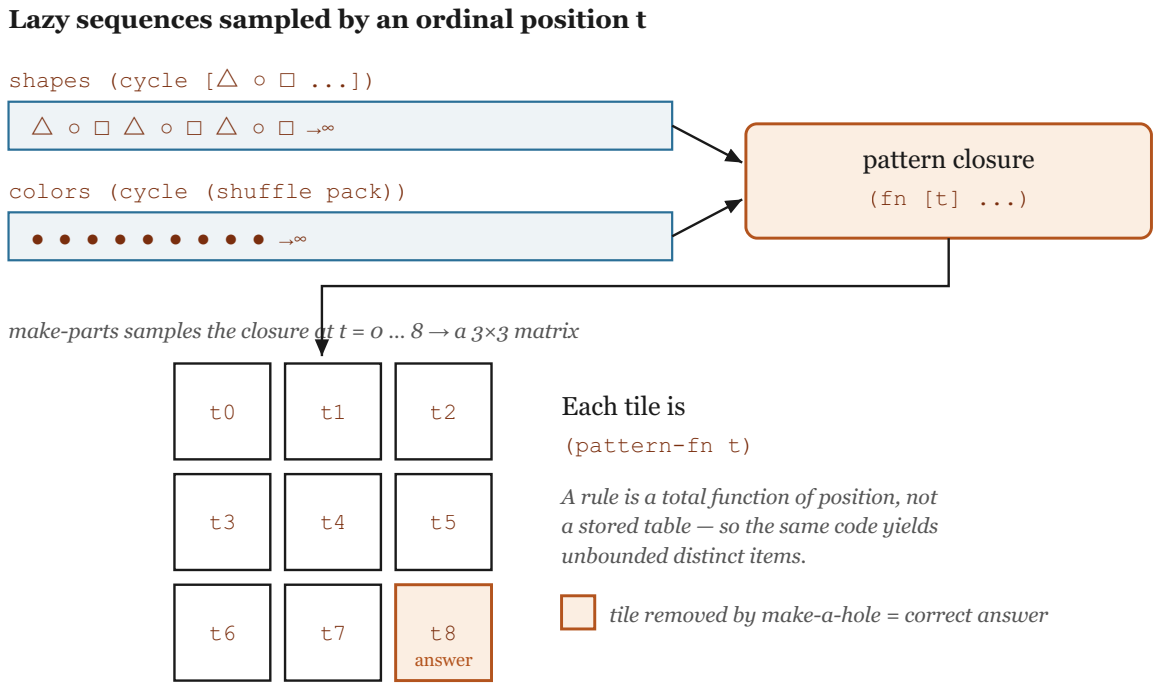


Figure 2. A rule is a total function of position. Independent lazy sequences (shape, colour, …) are sampled by `(fn [t])` at `t = 0…8`; `make-a-hole` then removes one tile to serve as the answer.

Other families vary the mapping while keeping the shape. The matchstick progression (`progressive_matrix_matchsticks.cljs`) builds three cyclic symbol sequences of *different lengths* and *different starting offsets*, one per drawing "level", so the three strokes of a glyph progress out of phase:

```
(let [level-seqs    (map #(take (u/srand-int (+ 1 %) 5) (nth seqs %)) levels)
      level-offsets (map #(rand-int (count %)) level-seqs)
      get-config    (fn [lvl t]
                      (let [seq (nth level-seqs lvl)
                            idx (mod (+ t (nth level-offsets lvl)) (count seq))]
                        (nth seq idx)))]
  (fn [t] (mapcat (fn [lvl] (get-lines (get-config lvl t) lvl ...)) levels)))
```

Listing 4 — Out-of-phase modular progressions per drawing level (`progressive_matrix_matchsticks.cljs`).

The grid-shapes family (`progressive_matrix_grid_shapes.cljs`) instead moves each object across a 4×4 grid by a signed modular step, `(mod (+ orig (* step t)) 16)`, a discrete analogue of translation. In every case the pattern is a closure of `t`; only the interpretation of `t` changes.

5. Answer perturbation

A generated matrix is only useful with a set of choices in which exactly one tile completes the rule and the rest are plausible but wrong. Two utilities frame the problem. `make-a-hole` removes a random question tile and returns it as the answer:

```
(defn make-a-hole [parts]
  (let [hole-key (rand-nth (keys parts))]
    [(assoc parts hole-key [{:shape :hole}]) (hole-key parts)]))
```

Listing 5 — Extracting the correct answer (`utils.cljc`).

Distractors are then produced by one of three strategies (Figure 3), chosen to suit the family's structure.

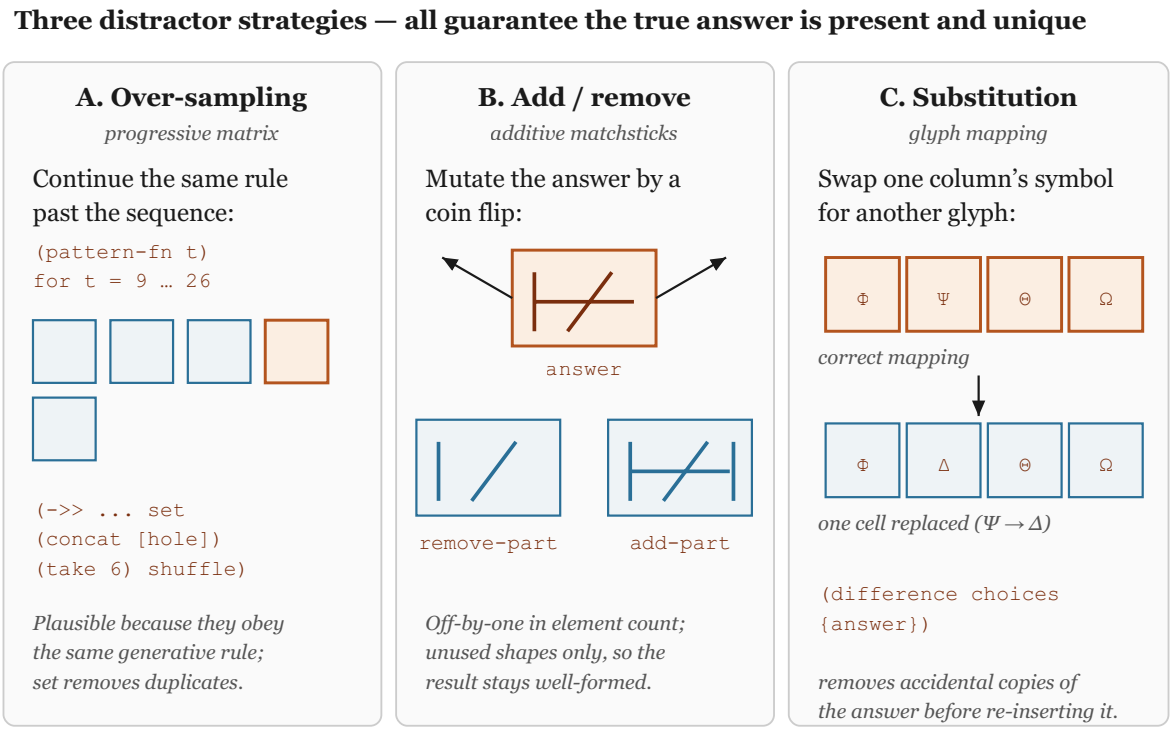


Figure 3. The three distractor strategies. All deduplicate via a Clojure `set` and guarantee that the true answer is present exactly once.

Strategy A — over-sampling the rule. The cleanest source of a plausible-but-wrong tile is the *same rule at a position that is not the answer*. `gen-choices` samples the pattern closure well past the nine visible cells (`SEQ-MAX = 27`), takes distinct results, injects the true answer, caps the set, and shuffles:

```
(defn gen-choices [pattern-fn hole choices-len seq-max]
  (let [choices (->> (make-parts seq-max pattern-fn "c")
                     vals
                     (take (dec choices-len))
                     (concat [hole])
                     set          ; deduplicate
                     (take choices-len)
                     vec
                     shuffle)]
    (make-parts (count choices) (fn [t] (nth choices t)) "c")))
```

Listing 6 — Over-sampling distractors (`utils.cljc`).

Because every distractor obeys the generative rule at *some* position, all options share the item's visual vocabulary; the `set` removes accidental duplicates, including any that coincide with the answer, and the answer is guaranteed present because it is `concat`-ed in before truncation.

Strategy B — structural add/remove. For the additive matchstick family, where a cell is a *set* of line segments, a natural distractor is the correct figure with one segment too many or too few. A coin flip picks the direction; the added segment is drawn only from shapes not already present, keeping the result well-formed:

```
(let [add-part    (fn [] ; add an unused line segment at a random level
                    (let [lvl (rand-int 3)
                          used (set (map :shape-code (filter #(= (:lvl %) lvl)
parts)))
                          available (cset/difference (set SHAPES) used)]
                      (conj-and-sort parts (make-line (first available) lvl))))
      remove-part (fn [] (-> parts shuffle drop-last sort-parts vec))
      trans-fn    (fn [_] (if (u/coin-flip) (remove-part) (add-part)))]
  (->> (map trans-fn (range 12)) shuffle (take (dec CHOICES-LEN))
       (concat [parts]) set (take CHOICES-LEN) vec shuffle))
```

Listing 7 — Add/remove perturbation, abridged (`additive_matrix_matchsticks.cljs`).

Strategy C — symbol substitution. In the glyph-mapping task the answer is a row of four symbols; a distractor replaces one column's symbol with a different glyph. The correct row is removed from the candidate pool by set difference before being re-inserted once, so it can never appear twice:

```
(defn gen-choices [part-answer syms]
  (let [swap-fn (fn [part sym] (assoc part :text sym))
        choices (reduce (fn [col sym]
                          (conj col (update-in part-answer
                                               [(-> (range 4) shuffle first)]
                                               #(swap-fn % sym))))
                        [] syms)
        choices (cset/difference (set choices) (set [part-answer]))]
    (->> choices (take 5) (concat [part-answer]) shuffle vec ...)))
```

Listing 8 — Substitution perturbation (`glyph_mapping.cljs`).

Across all three, two invariants hold by construction: the answer set is *deduplicated* (via `set`) and the *correct answer is always a member*. These are the logical preconditions for a single-solution item; the next section enforces the structural ones.

6. Logical validation

Validation in this system means **solution-integrity**: every emitted item must conform to a precise structural schema, carry only known keys, and be safe to render. We use Malli [8], whose schemas are ordinary data and can therefore be *built at run time from the item itself*.

The base document schema is fixed, but the per-tile content is not—different families populate `:part-infos` with different key sets. `get-eff-schema` therefore folds the item's own `:part-keys` into an *effective* schema, one closed map entry per tile:

```
(def SSDataSchema
  [:map {:closed true}
   [:id keyword?] [:version int?] [:uuid uuid?] [:name string?]
   [:generated-at any?] [:part-keys vector?]
   [:part-answer {:optional true} vector?] [:part-infos map?]])

(defn get-eff-schema [schema-part-infos schema-part-answer config]
  (let [part-infos-schema (reduce #(conj % [%2 schema-part-infos])
                                  [] (:part-keys config))]
    (-> SSDataSchema
        (mu/assoc :part-infos (into [:map] part-infos-schema))
        (mu/assoc :part-answer schema-part-answer))))
```

Listing 9 — A schema computed from the item's keys (`ssdata.cljs`).

Validation then both *checks* and *normalises*. `is-config-valid?` reports a humanised explanation on failure; `cleanup-progressive-matrix-config` decodes the config through a `strip-extra-keys-transformer`, discarding any field the schema does not declare, so intermediate bookkeeping keys used during generation (for example `:order`, `:lvl`, `:id` on matchstick segments) never reach the renderer:

```
(defn is-config-valid? [schema-part-infos schema-part-answer config]
  (let [eff-schema (get-eff-schema schema-part-infos schema-part-answer config)
        is-valid?  (m/validate eff-schema config)]
    (when-not (true? is-valid?)
      (prn "Config invalid" (-> eff-schema (m/explain config) (me/humanize))))
    is-valid?))

(defn cleanup-progressive-matrix-config [config]
  (let [eff-schema (get-eff-schema SSProgressiveMatrixDataSchema
                                   SSProgressiveMatrixDataSchema config)]
    (m/decode eff-schema config mt/strip-extra-keys-transformer)))
```

Listing 10 — Validate-and-normalise (`ssdata.cljs`).

The `{:closed true}` maps are the crux: a closed map *rejects* unexpected keys rather than ignoring them, turning a silent rendering bug (an unhandled shape field) into an explicit validation failure. Figure 4 shows the flow, including the humanised-error branch used during development.

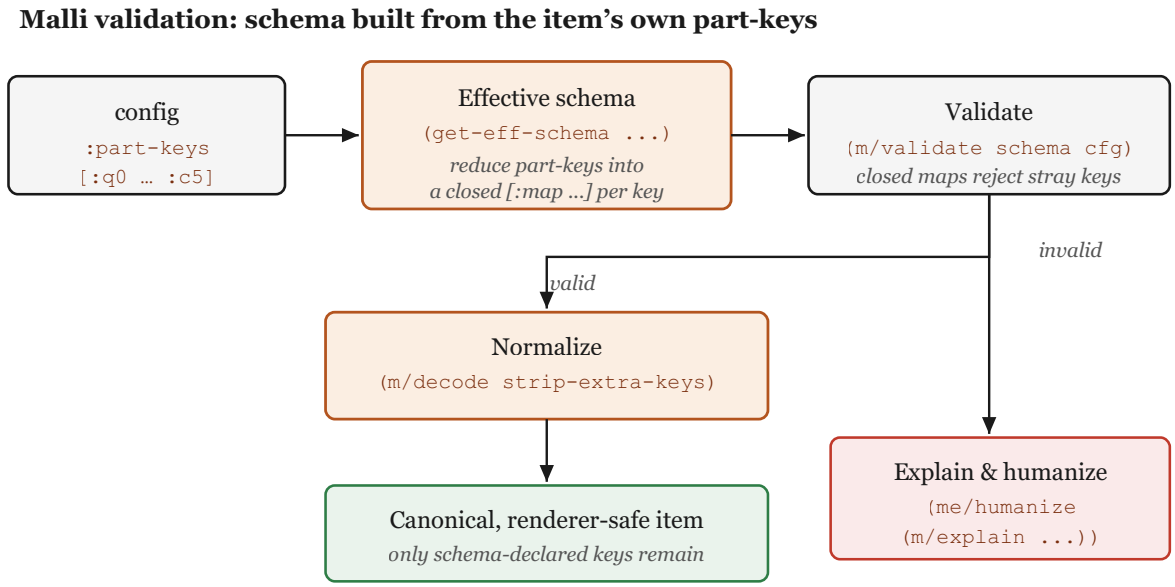


Figure 4. Validation builds an effective schema from `:part-keys`, checks the config against closed maps, and normalises valid items by stripping undeclared keys; invalid items are explained and humanised.

Distinct per-family content schemas (`SSProgressiveMatrixDataSchema`, `SSInspectionTimeDataSchema`, `SSCistercianNumeralDataSchema`) capture the shape grammar of each item type—e.g. that a progressive-matrix tile is a vector of maps each having a `:shape` keyword and optional `:pos`, `:size`, and colour fields—so a malformed tile from any generator is caught by the same machinery.

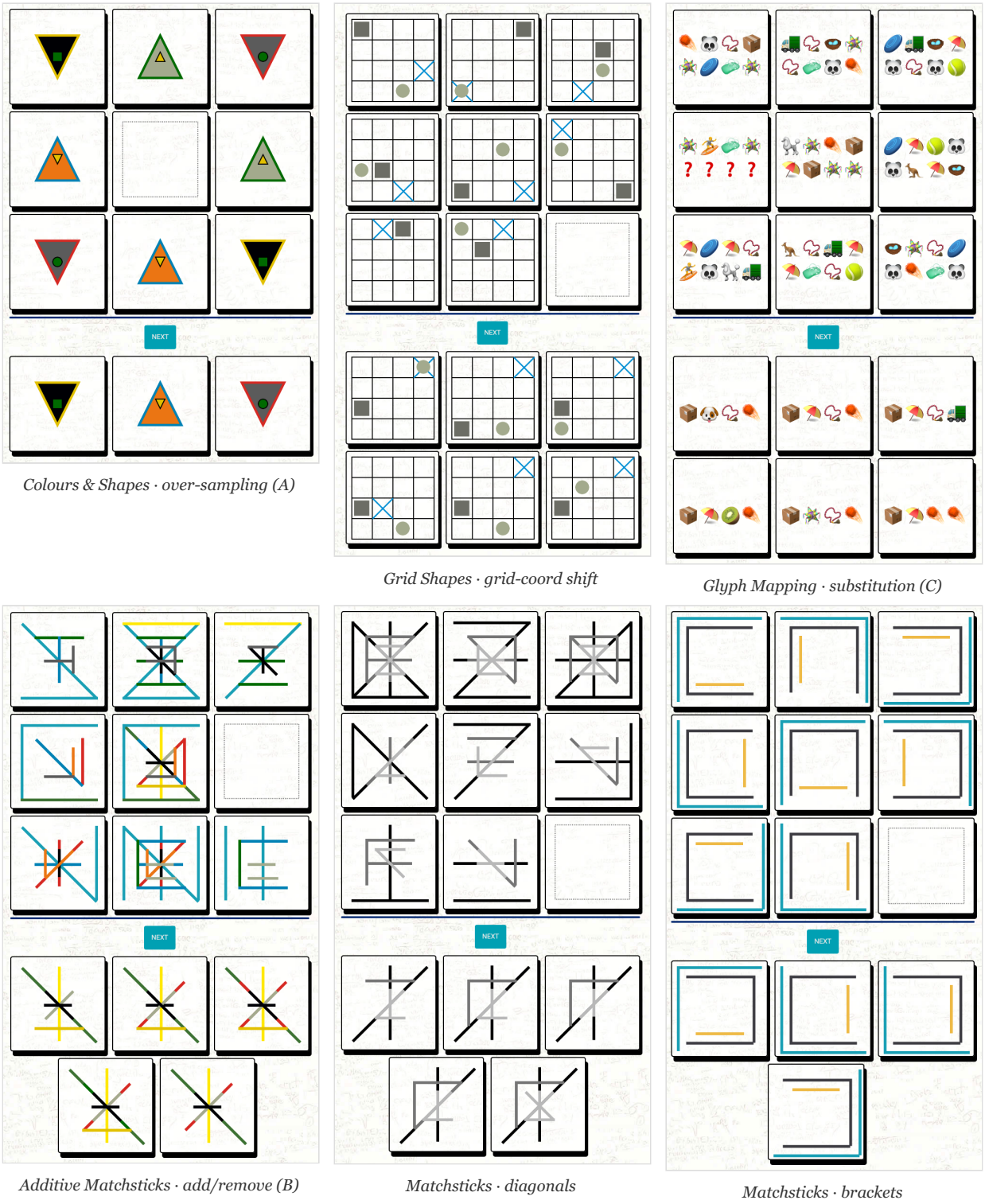
7. The generator taxonomy

Nine generator families are in production. They divide into *matrix* families (3×3 grids with derived multiple-choice answers) and *single-stimulus* families (one figure or a short arithmetic prompt with a computed answer). Table 1 summarises them; the three matrix families in bold are the case studies above.

Generator	Item form	Rule / transformation	Perturbation	Schema
progressive-matrix-colors-and-shapes	3×3 matrix	Per-channel cyclic shape & colour progression	Over-sampling (A)	ProgressiveMatrix
additive-matrix-matchsticks	3×3 matrix	Set union of matchstick segments across series	Add/remove (B)	ProgressiveMatrix
glyph-mapping	4-column mapping	Parent→child symbol association	Substitution (C)	(custom)
progressive-matrix-matchsticks	3×3 matrix	Out-of-phase modular symbol sequences per level	Over-sampling (A)	ProgressiveMatrix
progressive-matrix-grid-shapes	3×3 matrix	Objects translated on a 4×4 grid (mod 16)	Grid-coord shift; over-sampling fallback	ProgressiveMatrix
cistercian-numeral	Single glyph	Encode an integer 0–9999 as a Cistercian numeral	— (free response)	CistercianNumeral
inspection-time	Two hooks	Müller-Lyer-style length discrimination	— (binary left/right)	InspectionTime
percentage-of	Text prompt	"X is Y% of what number?" with integer solution	— (computed)	(plain map)
missing-average	Text prompt	Recover the 4th value given a target mean	— (computed)	(plain map)

Table 1. The nine generator families. "Perturbation" letters refer to the strategies of Section 6. The last two families return plain maps and bypass the Malli layer, reflecting their simpler, non-figural output.

Figure 5 makes the pipeline concrete with six items drawn straight from the running application. Each panel shows a complete generated problem: a 3×3 stem with the bottom-right cell withheld (dotted), above the strip of derived answer choices. Together they exhibit five of the matrix families in Table 1 and all three perturbation strategies of Section 6.



Colours & Shapes · over-sampling (A)

Grid Shapes · grid-coord shift

Glyph Mapping · substitution (C)

Additive Matchsticks · add/remove (B)

Matchsticks · diagonals

Matchsticks · brackets

Figure 5. Real items produced by the ClojureScript generator, as rendered in the *super-sharp* web application. Each panel is one complete problem—the 3×3 stem (bottom-right cell withheld) above its answer choices. The colours-and-shapes and grid-shapes panels progress figures across cells; the glyph-mapping panel maps parent symbols to child symbols (its withheld row marked `???`); the three matchstick panels build figures from coloured line segments.

8. Limitations and future work

Validation is logical, not psychometric. The Malli layer guarantees that items are *well-formed* and *single-solution by construction*, but it says nothing about their *difficulty*. Unlike

difficulty-calibrated generators [5, 6] or the empirically validated system of Wang and Su [4], *super-sharp* has not yet been subjected to a human study or an item-response-theory analysis. Estimating item difficulty from generative parameters (number of rules, motif period, distractor distance) and validating against response data is the central next step.

Reproducibility. Generation depends on Clojure's `rand`/`shuffle`; despite its name, the helper `srand-int` is not seeded (`(+ s (rand-int (- e s)))`). There is consequently no way to regenerate a specific item from a seed. A seeded PRNG threaded through the generators would make items addressable and test sets reproducible—a prerequisite for the difficulty study above.

Distractor quality. Over-sampling assumes that a later position of the rule is a *good* wrong answer; for short-period motifs it can occasionally coincide with the answer's visual near-neighbour, and while the `set` prevents exact duplicates, it does not measure semantic distance between distractors. A model of distractor plausibility would let the system target a desired discrimination.

9. Conclusion

We have described a small, uniform architecture for generating IQ-style nonverbal reasoning items. Its leverage comes from three functional-programming commitments: rules expressed as pure closures over lazy sequences, so an unbounded item space follows from a few lines of code; distractors *derived* from the same rules by three composable perturbation strategies that preserve a unique answer; and solution-integrity enforced by schemas *computed from each item's own structure*. The result cleanly separates what an item *is* from how it is drawn, and turns a class of would-be rendering bugs into explicit, humanised validation failures. The open frontier is empirical: connecting these generative parameters to measured difficulty, so that generation delivers not only well-formed items but psychometrically calibrated ones.

References

1. Raven, J. C. (1938). *Progressive Matrices: A Perceptual Test of Intelligence*. London: H. K. Lewis.
2. Raven, J. (2000). The Raven's Progressive Matrices: Change and stability over culture and time. *Cognitive Psychology*, 41(1), 1–48.
3. Carpenter, P. A., Just, M. A., & Shell, P. (1990). What one intelligence test measures: A theoretical account of the processing in the Raven Progressive Matrices Test. *Psychological Review*, 97(3), 404–431.
4. Wang, K., & Su, Z. (2015). Automatic Generation of Raven's Progressive Matrices. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI-15)*. AAAI Press.
5. Matzen, L. E., Benz, Z. O., Dixon, K. R., Posey, J., Kroger, J. K., & Speed, A. E. (2010). Recreating Raven's: Software for systematically generating large numbers of Raven-like matrix problems with normed properties. *Behavior Research Methods*, 42(2), 525–541.
6. Embretson, S. E. (1998). A cognitive design system approach to generating valid tests: Application to abstract reasoning. *Psychological Methods*, 3(3), 380–396.
7. Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3), 172–186.
8. Metosin. (2019–). *malli: Data-driven schemas for Clojure/Script*. https://github.com/metosin/malli
9. Hickey, R. (2020). A history of Clojure. *Proceedings of the ACM on Programming Languages*, 4(HOPL), Article 71.