Eight Experiments: The English Context Neuron

v2 (iterated) — revised after MJC commentary

Claude and MJC — February 24, 2026

These eight experiments trace a single thread: discovering the English context neuron from the data, defining subtraction in the UM, and measuring whether it compresses. v2 restates each experiment in UM-native terms, connects to $E \to \mathbb{N} \to \mathbb{Q}$, and incorporates MJC's commentary nearly verbatim.

1. Experiment 1: Output ES Factoring

1.1. v1 summary

Inspect per-source-event entry counts in each LPP. Collect the distribution of fan-out numbers. Look for clustering.

1.2. MJC commentary

The question is a bit dumb — we don't need scare quotes around "factor" because it's literal factoring in the UM algebra.

The experiment is a good idea, however the implementation is very stupid. What we have in the UM once the LPPs are trained is the factors, via pattern chains onto the output space E_O. All you have to do is literally read the support out of the LPP at each point and then take the softmax to get a probability distribution and then go to the next step in each chain until you reach the ocean (O, the output space in $I \times H \times O$).

Good idea: restate it in UM terms. Principle: see if there's a UM-native way to formulate the information you're seeking, and if you can collect the data by setting up an appropriate LPP. The experiment design started from "for every LPP" and continues by describing what an LPP gives you for free.

On v2: "chain-analysis" is a bit of a generic term for analysis of chains terminating at O, but otherwise the experimental design is good.

1.3. v2: UM-native formulation

The factoring structure is already in the trained model. For each LPP $L : E_s \to E_O$ in the trained bigram:

1. For each source event $e \in E_s$ with support > 0, read L's entries: $\{(e, o, w) : o \in E_O\}$.
2. Take the softmax: $p(o \mid e) = 2^{w_{e,o}} / \sum_{o'} 2^{w_{e,o'}}$.
3. This IS the factor: the distribution $p(\cdot \mid e)$ is the contribution of event e to the output.

For pattern chains (e.g., $I \to H \to O$), follow the chain: read the first LPP's distribution, propagate via max-min through the second, read the output distribution at the end. Each chain gives a different factoring of the output.

What to measure: For each chain terminating at O, compute its output distribution and its entropy $H(O \mid \text{chain})$. Chains with lower entropy = sharper factors = more informative contexts. This is exactly the sharpest-LPP metric, now understood as measuring which factor carries the most information.

$E \to \mathbb{N}$ **connection**: Each LPP entry (e_s, e_o, w) is a term in the prime encoding. The source event selects a prime power; the target is in E_O. The entry count per source event = the number of non-zero exponents for that prime in the product polynomial. Factoring the output distribution = extracting these exponents.

What we build: `umr output-factoring model.sn` — for each LPP chain to O, report the per-source-event output distributions and their entropies. No new UM structure; this reads the existing model.

2. Experiment 2: English Orthography Failure Map

2.1. v1 summary

Run surprise analysis with per-class breakdown: (a) XML tag, (b) entity, (c) word boundary, (d) lowercase, (e) uppercase, (f) digit, (g) punctuation, (h) other. Report mean surprise per class.

2.2. MJC commentary

Good idea, now restate it in UM terms. We already store a memory trace, we know how big it is, we don't need `umr surprise` anymore. The 8 classes a–h are good proto-context neurons. Especially the digit one is built to last.

However, the experimental design is stupid for the same reasons as Exp. 1: running a separate surprise analysis tool is ad-hoc when the UM already produces per-position distributions.

On v2: This is good. We can actually run this in parallel with other experiments by extending the UMR with a single simple feature: an LPP that we run learning on but that is not used for inference — it doesn't add support in any direction in the forward pass.

2.3. v2: Observation-only LPP

The 8 proto-context neurons should be defined as events in a new ES. The key infrastructure addition: an **observation-only LPP** — an LPP that participates in ω_0 (learning) but is skipped during the forward pass f. It collects joint-event statistics without influencing the model's predictions.

This requires a UMR spec extension. Proposed SN syntax:

```
LPP "proto_context" "byte_output" OBSERVE_ONLY
```

The `OBSERVE_ONLY` annotation means:

- During the forward pass: this LPP is *skipped*. It does not propagate support from source to target. The model's predictions are unchanged.
- During learning (ω_0): this LPP *does* observe joint events. When both source and target events have support > 0, the LPP entry is log-stochastically incremented as usual.
- After training: the LPP entries contain joint-event statistics that can be read out as per-context output distributions.

This lets us run the observation-only LPP *in parallel* with the production model. The proto-context ES fires based on byte class (absolute patterns from I), the observation-only LPP silently accumulates context×output statistics, and the actual predictions come from the existing bigram LPPs as before.

The 8 proto-context events:

```
ES "proto_context" 8
  "The current byte is inside an XML tag."    0.
  "The current byte is inside an XML entity." 0.
  "The current byte is at a word boundary."   0.
```

```
"The current byte is a lowercase letter."   0.
"The current byte is an uppercase letter."  0.
"The current byte is a digit."              0.
"The current byte is punctuation."          0.
"The current byte is none of the above."    0.
```

Spec proposal (for discussion — changes to #umr_spec):

- §3 (LPPs): add `OBSERVE_ONLY` flag. "An observe-only LPP participates in ω_0 but is excluded from f. It is a passive instrument: it measures joint events without affecting inference."
- §10 (SN format): `LPP "from" "to" OBSERVE_ONLY.` as valid syntax.
- `um_forward()`: skip LPPs where `observe_only == 1`.
- `um_lpp_observe()`: process observe-only LPPs normally.
- JS viewer: parse and display observe-only LPPs (greyed out in the forward pass visualization, shown in the learning visualization).

What to measure: After training with the observation-only LPP, read out $p(o \mid c) = 2^{w_{c,o}} / \sum 2^{w_{c,o'}}$ for each proto-context c. Report entropy per context. The digit context neuron "is built to last" (MJC) because digits have a radically different distribution, persisting at every level of the factor tower.

3. Experiment 3: The English Context Neuron (SN)

3.1. v1 summary

Define a 2-event context ES (English/Other). Activation: two consecutive lowercase letters → English; < → Other. Add context→output LPP. Measure bpc gain.

3.2. MJC commentary

Question is not a good one, because the answer is obviously and trivially yes. Why are two lowercase letters required before you start?

This is a pointless experiment to run but an instructional one to design. Write it in SN terms.

3.3. v2: SN specification

The English context neuron as a complete SN model:

```
ES "byte_input"  256
ES "byte_output" 256 OUTPUT
ES "byte_prev"   256
ES "context"       2
  "We are in English text context." 0.
  "We are in non-English context."  0.

SHIFT "byte_output" "byte_prev"

# Activation: absolute patterns byte_input -> context
# byte_input[97..122] ('a'..'z') -> context[0] ("English")
97 512 255.
98 512 255.
...
122 512 255.
# All other byte_input events -> context[1] ("non-English")
0 513 255.
...
96 513 255.
123 513 255.
...
255 513 255.

LPP "byte_prev" "byte_output"
LPP "context"   "byte_output"
```

(512, 513 = context.base + 0, 1. The ... elides the remaining patterns in each range.)

The context ES has two events. The 26 lowercase input bytes activate "English" via absolute patterns at weight 255; the remaining 230 input bytes activate "non-English". The context→output LPP learns two distributions: $p(o \mid \text{English})$ and $p(o \mid \text{non-English})$.

Why trivially yes: The English-context LPP will learn approximately the English letter frequency table when English is active. This is guaranteed to have lower entropy than the unconditional distribution (which mixes English with XML, digits, etc.). So the answer is yes, the context neuron helps. The question is not *whether* but *how much*, and how to formalize the "subtraction" that this context performs.

Instructional value: Writing the SN forces us to specify exactly what events exist, what LPPs connect them, and what the forward pass computes. The SN IS the experiment.

4. Experiment 4: Log-Stochastic Subtraction

4.1. v1 summary

Formalize "subtracting English from the output" as sharpest-LPP selection between context-specific and generic distributions. Build a per-position subtraction map.

4.2. MJC commentary

These questions have known answers; the task is to find references in the active papers. It is literally log-stochastic subtraction. Connect this to $E \to \mathbb{N} \to \mathbb{Q}$.

On v2: this is incomplete. Log-stochastic subtraction is an actual subtraction mechanism on log-support values. You know log-stochastic increment and should be able to derive LS addition and subtraction.

On v2 iterated: the actual idea is that we can perform model surgery and do the LS subtraction on the pattern space, and then the residual actually becomes the new marginal. When the English context is active, the LPP from "We are in English" onto O is added to the residual. What we have done is taken the LPP from True to O and split it into the part where E_{lang} is English and the remaining part — we factored it, took the quotient by one event in the new factor, and then took the difference (in the pattern space). Now we have an LPP from English and from Other both going onto O. This is something we could do, but it's also equivalent to just setting up the language ES and retraining the model on the memory trace to learn the two marginals separately. In any case this operation also lets us set up any kind of Bayesian marginalization properly because we have trivial identity on the microstates of the system: we know exactly what we are double-counting.

4.3. v2: Log-Stochastic Algebra (LSA)

We define **Log-Stochastic Algebra** (LSA) as the arithmetic of log-support values. Reference: CMP paper (toc.tex), section on log-stochastic arithmetic.

4.3.1 The three LSA operations

LS increment: Given support s, increment with probability 2^{-s}. Adding 1 observation to $\sim 2^s$ crosses the next power of 2 with probability $\sim 2^{-s}$.

LS addition ($\oplus$): To add s_b to s_a (where $s_b \leq s_a$), let $d = s_a - s_b$. Increment s_a with probability 2^{-d}.

Rationale (CMP): "if we add support values of 2 and 3, we should increment the larger value (the 3) with probability 1/2, because going from 3 to 4 means adding linear support of 8, and if, every time we should add 4, we half the time add 8 and half the time add nothing, we will come out alright."

LS subtraction ($\ominus$): To subtract s_b from s_a (where $s_b \leq s_a$), let $d = s_a - s_b$. *Decrement* s_a with probability 2^{-d}. Symmetric with addition: same probability, opposite direction.

Special cases: $s_b = s_a$ ($d = 0$): always decrement. $s_b = 0$: almost never decrement. $s_b > s_a$: clamp to 0.

4.3.2 Model surgery: factoring the True$\rightarrow O$ LPP

The key operation is **model surgery on the pattern space**. We start with:

```
ES "envelope" 1
  "True." 0.
ES "byte_output" 256 OUTPUT
LPP "envelope" "byte_output"    -- the unigram marginal
```

The True$\rightarrow O$ LPP has 256 entries, one per output byte b, with support $s_U(b)$ — the unigram marginal over all contexts.

Now introduce E_{lang} with events "English" and "Other." Suppose we have (from an observation-only LPP, or from retraining) the English-context distribution $s_{\mathcal{E}}(b)$ — the support for each output byte restricted to positions where English is active.

Surgery: For each output byte b, LS-subtract the English contribution from the marginal:

$$s_R(b) = s_U(b) \ominus s_{\mathcal{E}}(b) \qquad \forall\, b \in E_O$$

The residual $s_R(b)$ **becomes the new marginal** — it replaces the True$\rightarrow O$ LPP. The English contribution becomes its own LPP: English$\rightarrow O$ with support $s_{\mathcal{E}}(b)$.

After surgery:

```
ES "envelope" 1
  "True." 0.
ES "context" 2
  "We are in English text context." 0.
  "We are in non-English context."  0.
ES "byte_output" 256 OUTPUT
LPP "context" "byte_output"        -- English: s_E(b), Other: s_R(b)
```

We have split one LPP (True$\rightarrow O$) into two (English$\rightarrow O$ and Other$\rightarrow O$, packaged as one LPP from the context ES). The old True$\rightarrow O$ LPP is gone — the residual has absorbed it.

At inference: when the English context fires, its LPP adds $s_{\mathcal{E}}(b)$ onto O. When "Other" fires, $s_R(b)$ goes onto O. Both are sharper than the original $s_U(b)$ because each explains only its own portion of the data.

Verification: $s_{\mathcal{E}}(b) \oplus s_R(b) \approx s_U(b)$ for all b. The factorization is lossless up to LS stochastic error.

4.3.3 Equivalence to retraining

This surgery is equivalent to: add E_{lang}, keep the two LPPs empty, and retrain the model on the memory trace. During retraining, the English$\rightarrow O$ LPP observes joint events only at English positions, learning $s_{\mathcal{E}}(b)$. The Other$\rightarrow O$ LPP observes at non-English positions, learning $s_R(b)$.

The surgery gives us the same result without retraining. This matters because it shows the factorization is algebraically exact: we are splitting counts that we already have, not re-estimating them. There is no double-counting because we have **trivial identity on the microstates** — every position in the data is either English or Other, and we know which. The LS subtraction respects this partition exactly.

4.3.4 Bayesian marginalization

This generalizes. For *any* new factor E_{new} that partitions the data positions into classes, we can:

1. Train (or observe) $s_c(b)$ for each class $c \in E_{\text{new}}$.
2. LS-subtract each class's contribution from the marginal in sequence.
3. Replace the single True$\to O$ LPP with per-class LPPs.

This is proper Bayesian marginalization: we decompose $p(b) = \sum_c p(c) \cdot p(b \mid c)$ where the sum becomes LS addition and the per-class conditionals become separate LPPs. The double-counting problem that plagues naïve model combination vanishes because we have the microstate identity: we know exactly which positions belong to which class, so no observation is counted in two classes.

4.3.5 $E \to \mathbb{N} \to \mathbb{Q}$ connection

In the prime encoding, assign prime p_b to each output byte b. Then:

$$N_U = \prod_b p_b^{s_U(b)} \qquad \text{(unigram marginal as integer)}$$

$$N_{\mathcal{E}} = \prod_b p_b^{s_{\mathcal{E}}(b)} \qquad \text{(English factor)}$$

$$N_R = N_U \;/\; N_{\mathcal{E}} = \prod_b p_b^{s_U(b) - s_{\mathcal{E}}(b)} \qquad \text{(residual = integer division)}$$

LS subtraction on support values = division in $\mathbb{N}$. LS addition = multiplication in $\mathbb{N}$. The surgery is: factor N_U into $N_{\mathcal{E}} \times N_R$, then use the factors separately.

The quotient $\mathbb{Q}$: define $t_1 \sim t_2$ iff $N_{t_1}/N_{\mathcal{E}} = N_{t_2}/N_{\mathcal{E}}$. Two positions are equivalent modulo English if they have the same residual. The equivalence classes of $\mathbb{Q}$ are the contexts that the *next* neuron must distinguish.

4.3.6 Classes of events

The LS subtraction partitions E_O into classes:

- **English-dominated**: $s_{\mathcal{E}}(b) \approx s_U(b)$, so $s_R(b) \approx 0$. These bytes (`e`, `t`, `a`, `o`, ...) are almost entirely explained by English. They nearly vanish from the residual.
- **English-independent**: $s_{\mathcal{E}}(b) \approx 0$, so $s_R(b) \approx s_U(b)$. These bytes (`<`, `>`, digits, `=`, `"`) are untouched by the English context.
- **Mixed**: $0 < s_{\mathcal{E}}(b) < s_U(b)$, so s_R is intermediate. These bytes (`.`, `,`, `-`, space) appear in both English and non-English contexts.

These classes are discoverable from the trained LPP support values, not imposed externally.

4.3.7 LSA operations in the UMR CLI

The UMR should expose LSA as first-class operations on LPP and ES objects, giving visibility into the algebra from the command line. Proposed commands:

```
umr lsa-subtract model.sn <lpp> <lpp>
```

LS-subtract one LPP's support values from another's, entry by entry on the shared target ES. Reports the residual distribution (per-event support before and after). This is the core inspection tool: given the True$\rightarrow O$ LPP and a trained English$\rightarrow O$ LPP, show what's left.

```
umr lsa-add model.sn <lpp> <lpp>
```

LS-add two LPPs' support values on their shared target ES. Verification tool: English $\oplus$ residual should reconstruct the original marginal.

```
umr lsa-factor model.sn <lpp> <es>
```

The full surgery: factor an LPP by a context ES. For each event c in the context ES, extract $s_c(b)$ from the LPP (restricted to positions where c was active), LS-subtract from the marginal, and write a new SN with per-class LPPs replacing the original.

```
umr lsa-dist model.sn <lpp|es> [event]
```

Read out the softmax distribution for a given LPP (optionally restricted to a single source event) or an ES's current support. Reports entropy, top-k events, and support values. The basic "look at this object" command.

These extend the UMR spec with an **algebraic operations** section (§TBD): LSA as operations on the pattern space P, defined on support values in LPP entries and ES events. The operations are stochastic (they use a PRNG) but correct in expectation. Multiple applications converge.

What we build: LSA primitives ($\oplus$, $\ominus$) in `#umr_core`. The four CLI commands above. An algebraic ops section in `#umr_spec`. These are built when we run the experiment, not before.

5. Experiment 5: Word Events and the Frequency Table

5.1. v1 summary

Chain: English context → word events → byte output. Top 100 words. Measure whether the word-mediated chain recovers the lowercase frequency table.

5.2. MJC commentary

This is getting at the definition. English is DEFINED AS runs of English words. The orthography says how to map those onto bytes. We have the orthography and we're getting at the lexicon.

This is also not an experiment but part of the definition that you have to work out. A first-order Markov approximation of English is word frequencies, which we are in a position to have IF we have the context byte, which means the words and the English are co-extensive in the dataset.

This is all definitional, but you should make it clearly so via $E \to \mathbb{N} \to \mathbb{Q}$ as well.

However, §5.2 is a key moment — the first experiment that's well described and worth doing. Starting with the top 100 words and doing this manually lets us state what we want the SN to look like before and after, and then this all becomes much more practical to learn automatically the next N times we do it.

However, the words need to map onto the bigram and trigram tables, not just O.

All of this is moving towards being able to use simple algebraic expressions to describe models. The frequencies of those 100 words can be measured (you can do this with a shell one-liner if you like). You can then multiply and get the exact output distribution. However, what we are interested in is the orthography, so this maps onto more of the output tower.

On v2: this is good.

5.3. v2: Definitions and the key experiment

Definition: English text = runs of English words. A word is a sequence of bytes between reset signals (space, newline). The English context neuron fires during these runs. The words and the English context are co-extensive in the dataset.

First-order approximation: Given word frequencies $f(w)$ for the top 100 words, the expected letter distribution at any position within English text is:

$$p(b) = \sum_{w} f(w) \cdot p(b \mid w)$$

where $p(b \mid w)$ is the per-byte distribution when spelling word w. This should approximate the empirical lowercase frequency table.

$E \to \mathbb{N}$ **connection**: Each word event w is a prime. Its frequency $f(w)$ gives its support: $s(w) \approx \log_2 f(w)$. The product $\prod p_w^{s(w)}$ over all word events is the integer encoding of the lexicon. Factoring this integer recovers word frequencies. The quotient $\mathbb{Q}$ over word events groups positions by which word they belong to.

The key experiment (§5.2):

1. Measure the top 100 word frequencies in enwik9 (shell one-liner: extract space-delimited tokens, count, sort).
2. For each word, write its SN: the word event, LPPs to bigram and trigram tables (not just O directly), and the orthographic chain that spells it out.
3. Write the SN for the full model: 100 word events, context→word LPP with frequency-based support, word→bigram and word→trigram LPPs.
4. State what the SN looks like *before* training (architecture only, no learned weights).
5. State what the SN should look like *after* training (with expected support values).
6. Run the 3-pass pipeline. Compare actual SN to expected.

This is manual and definitional: we are writing down what the model *should* look like if the definitions are correct, then checking. This lets us build the automatic version next.

What we build: The 100-word SN model. The shell one-liner for word frequencies. The comparison tool (diff between expected and actual SN support values).

6. Experiment 6: Uppercase (P-Programming Exercise)

6.1. v1 summary

Add a third event (English_upper) to the context ES. Measure independent gain from separating uppercase.

6.2. MJC commentary

Interesting but not worth a separate experiment. This can literally be an entry-level P-programming problem: add an event for UPPERCASE and add an LPP for it and then you get your experimental data for free.

On v2: grade this as a programming exercise. Event names must be English declarative sentences. You may define a short name for each one or implicitly by context use a short name. Every atomic event must have a unique name; "Other." gets an F. The activation should be a P-program, not a prose description of what you want.

6.3. v2: P-programming exercise (graded)

Complete SN:

```
ES "byte_input"  256
ES "byte_output" 256 OUTPUT
ES "byte_prev"   256
ES "context" 3
  "The current text is English lowercase." 0.
  "The current text is English uppercase." 0.
  "The current text is not English."       0.

SHIFT "byte_output" "byte_prev"

# Activation: absolute patterns from byte_input to context.
# byte_input[97] = 'a', ..., byte_input[122] = 'z'
97 512 255.
98 512 255.
...
122 512 255.
# byte_input[65] = 'A', ..., byte_input[90] = 'Z'
65 513 255.
66 513 255.
...
90 513 255.
# All other byte_input events -> "not English"
0 514 255.
...
64 514 255.
91 514 255.
```

```
...
96 514 255.
123 514 255.
...
255 514 255.

LPP "byte_prev" "byte_output"
LPP "context"   "byte_output"
```

(Where 512, 513, 514 are the global indices of the three context events: context.base + 0, 1, 2. The `...` elides the remaining absolute patterns in each range.)

These are absolute patterns at weight 255: when byte_input[97] ("The input byte is 'a'.") fires, it sets context[512] ("The current text is English lowercase.") to 255 via the max-min forward pass. The 26 lowercase, 26 uppercase, and 204 remaining input bytes each map to exactly one context event. This is the P-program.

After training, the three per-context output distributions are the experimental data. No separate measurement tool. The LPP entries ARE the data.

Exercise: Write this SN, run it, read the trained SN. Report the three distributions and their entropies.

7. Experiment 7: Tag Open Turns English ON

7.1. v1 summary

"<" deactivates English context (switches to XML mode). Instrument context switches.

7.2. MJC commentary

Exactly backwards. The point is that "<" TURNS ENGLISH MODE ON, not OFF. That's because tag names are English words or close to them. The space ends the tag name. Of course we can have a different distribution on tag names, but it adds to the word distribution, it is not separate.

On v2: they are not sub-contexts of English either. They just re-use the word distribution and the tower below it (towards O).

7.3. v2: Corrected model

The v1 design was exactly wrong. XML tag names (`title`, `text`, `id`, `redirect`, `timestamp`) are English words or English-derived. After `<`, the model is still predicting English-like letter sequences. It is the *space* (or `>` or `/`) after the tag name that ends the English-like distribution.

The correct activation model:

- `<` → English mode stays ON (tag name is English-like).
- Space/`>`/`/` after `<` → context may shift (attribute values, tag close).
- Digits, `=`, `"` inside tags → non-English contexts.

Tag names re-use the word distribution and the entire tower below it toward O. They are not sub-contexts of English — they participate in the same word distribution, just with restricted vocabulary. The English context neuron fires for tag names exactly as it fires for prose words. The tag-name distribution *adds to* the word distribution; it is not separate.

The non-English contexts are: digits, entities (`&`), attribute values (quoted strings with mixed content), markup syntax (`</`, `/>`, `=`).

What we build: Correct the activation rules in the Exp. 3 SN. Verify that tag names are predicted well when English context is active. The experimental data is the per-position surprise inside `<...>`: if the English context helps here too, the model is correct.

8. Experiment 8: One-Context-at-a-Time Learning (Conclusion)

8.1. v1 summary

Sequential discovery: learn one context, subtract it, find the next from the residual. No algorithmic clustering.

8.2. MJC commentary

Yes, you described what we are doing. Not an experiment, but fine as a conclusion section.

8.3. v2: The protocol

This is the research protocol, not an experiment:

1. **Start**: unigram model, memory trace, baseline bpc.
2. **Define context**: from inspecting the model's failures (reading the trained LPP distributions, not running external scripts), hypothesize a context neuron.
3. **Add to SN**: write the context event and its LPPs in SN.
4. **Tick**: train the extended model on the memory trace. Write new trace.
5. **Measure**: new trace size vs old. If shorter, the context was real.
6. **Subtract**: the new context is now part of the model. Its contribution is "subtracted" (LS subtraction of explained distribution from the output).
7. **Repeat**: inspect the new residual, find the next context.

Each round adds one context neuron. The order should be:

1. English lowercase (largest signal, ∼60–70% of positions).
2. Digits ("built to last" — radically different distribution, persists at every level).
3. Uppercase English (sentence starts, proper nouns).
4. XML syntax (`/`, `>`, `=`, `"`).
5. Entities (`&`, `<`, etc.).

The cumulative effect: each context neuron explains a portion of the output distribution. Together, they partition the byte positions into classes with distinct distributions, achieving what the bigram model achieves through brute-force memorization but with meaningful, interpretable structure.

This is the Tock process: inspect the model, add structure (+E, +P), then Tick (retrain). The human provides the context hypothesis; the UM discovers the frequencies. Eventually, the UM should discover contexts too — but first we do it manually to understand the process.

Summary of changes from v1:

- Experiments restated in UM terms: read LPP distributions instead of running external analysis tools.
- Connected to $E \to \mathbb{N} \to \mathbb{Q}$: prime encoding, integer division as subtraction, quotient as equivalence class.
- Exp. 1: renamed command to `output-factoring`.
- Exp. 2: observation-only LPP proposed (learns but skips forward pass). Spec changes proposed for #umr_spec.
- Exp. 4: full derivation of LS subtraction from LS increment. Decrement s_a with probability $2^{-(s_a - s_b)}$.
- Exp. 6: event names as English declarative sentences. "Other." $\to$ "The current text is not English." Activation described declaratively.
- Exp. 7: corrected: `<` turns English ON. Tag names re-use the word distribution and tower toward O, not sub-contexts.
- Exp. 5.2 identified as the key experiment: top 100 words, manual SN, before/after comparison.
- Exp. 8 reframed as protocol (conclusion), not experiment.

The one experiment to run first: Experiment 5.2 — the 100-word SN model with manual frequency measurement and before/after SN comparison. This is concrete, testable, and directly advances the lexicon.

Parallel with Exp. 2: The observation-only LPP (once approved and implemented) lets us collect proto-context statistics during any training run without affecting results.