P-Programs:
Explicit Pattern Programs for the Extended Event Space

Claude and MJC

February 16, 2026

Abstract

We make the extended event space operational by writing explicit *P-programs*: compositions of patterns in SN concrete syntax that implement position tracking, letter accumulation, bag-of-letters recognition, and word identification. Each P-program is a small UM instance whose inner hidden events do not connect outside it—the self-similarity principle applied at the program level. We show that the accumulator is a tiny memory whose states biject onto its memory traces, formalizing the connection between accumulator events and the memory-trace framework of January 31. We give the abductive learning rule for online vocabulary growth: when surprise exceeds a threshold, the system abducts a new word event and records it as a memory trace for later ω-processing. We state and motivate the *safe-combination conjecture*: for every unsafe (correlated) evidence combination identified in [3], there exists a P-program refinement that makes it safe, with the $E \to N \to Q$ chain as the guide.

1 P-Programs

Definition 1 (P-program)**.** *A* P-program *is a finite collection of patterns $\Pi = \{(e_i, e_j, w_{ij})\} \subset E^2 \times T$ that together implement a deterministic or probabilistic computation. A P-program has:*

- ***Interface events****: events in I or O that connect the program to the rest of the model.*
- ***Internal events****: events in H that are used only within the program (no patterns connect them to events outside Π).*
- ***Deterministic patterns****: weight 255 (strength $s_{\max}$), representing logical implications that hold always.*
- ***Learned patterns****: weight < 255, representing statistical regularities counted from data.*

Proposition 2 (P-programs are inner UMs)**.** *A P-program Π with internal events H_Π is itself a UM instance $u_\Pi = (e_\Pi, t_\Pi, p_\Pi, f_\Pi, \omega_\Pi)$ nested inside the outer model. By the self-similarity principle [2], the internal events of Π are part of H from the outer perspective but form a complete $I_\Pi \times H_\Pi \times O_\Pi$ from the inner perspective.*

Remark 3 (Programs compose like procedures)**.** *UM instances compose under the product of their event spaces. A P-program Π_1 with output events O_1 can feed into a P-program Π_2 with input events $I_2 = O_1$. The composition $\Pi_2 \circ \Pi_1$ is itself a P-program. The maximum pattern depth inside a composed program is visible outside as* timing*—the circuit depth of the computation. This connection between pattern depth and circuit depth is already explicit in CMP [1].*

2 Program 1: The Position Counter

The in-word position counter is the simplest P-program: a deterministic counter that resets at word boundaries.

Definition 4 (Position counter P-program)**.** *Events:*

- *Interface input: byte events* $b \in \{0, \ldots, 255\}$.
- *Interface output / internal state: position events* pos_k *for* $k \in \{0, 1, \ldots, L_{\max}\}$.
- *Internal: word-boundary detector* wb.

Patterns in SN syntax:

“The input is ‘ ’.” → “Word boundary.”	*255*
“The input is “.” → “Word boundary.”	*255*
“The input is ‘,’.” → “Word boundary.”	*255*
“The input is 0x0A.” → “Word boundary.”	*255*
(one pattern per boundary character)	
“Word boundary.” → “Position is 0.”	*255*
“Position is 0.” → “Position is 1.”	*255*
“Position is 1.” → “Position is 2.”	*255*
$\vdots$	
“Position is $L_{\max} - 1$*.” → “Position is* $L_{\max}$*.”*	*255*
“Word boundary.” → “Position is 1. (reset)”	*255*

Remark 5 (Timing: the reset signal)**.** *The position counter needs an external reset signal: a word-boundary byte arriving on I. The word-boundary detector* wb *is an internal event that fires when the input byte is a boundary character. The reset pattern (wb → pos*$_0$*) has circuit depth 2 (byte → wb → pos*$_0$*); the increment patterns have depth 1 (pos*$_k$ *→ pos*$_{k+1}$*).*

The counter runs at the tick rate (one increment per byte). After a reset, it takes one tick to reach pos$_1$*, two ticks to reach pos*$_2$*, etc. This is the position counter that the RNN implements implicitly via* W_h *rotation [5].*

Proposition 6 (The position counter is a P-program with no learned patterns)**.** *All patterns in the position counter have weight 255. No counting (*ω_0*) is required. The program is pure logic, derivable from the definition of “word boundary” without seeing any data. A UM runner is free to short-circuit it.*

3 Program 2: The Letter Accumulator

Definition 7 (Letter accumulator P-program)**.** *Events:*

- *Interface input: byte events* $b \in \{0, \ldots, 255\}$*, position events* pos_k.
- *Internal state / interface output: accumulator events* acc_c *for each byte value c (“byte c has appeared in the current word”).*

Patterns in SN:

“The input is ‘t’.” → “Letter ‘t’ has appeared.”	*255*
“The input is ‘h’.” → “Letter ‘h’ has appeared.”	*255*
(one pattern per byte value: 256 patterns)	
“Word boundary.” → “Reset all accumulators.”	*255*

The accumulator event acc_c *is a* latch*: once set by seeing byte c, it remains set until the word-boundary reset. In the tropical semiring, a latch is a pattern from* acc_c *to itself with weight 255 (self-sustaining support), gated by the absence of a reset signal.*

Proposition 8 (Joint events in the accumulator)**.** *The accumulator naturally produces joint events:*

- *“Letter ‘t’ at position 1” is the joint event* $(\mathrm{acc}_t = 1) \wedge (\mathrm{pos} = 1)$*.*
- *“Letter ‘t’ at any position” is the marginal event* $\mathrm{acc}_t = 1$ *(the accumulator fires regardless of position).*
- *“Letters* $\{t, h\}$ *have appeared” is the conjunction* $\mathrm{acc}_t \wedge \mathrm{acc}_h$*.*

These accumulate into a joint event space: the product $\{0,1\}^{256} \times \{0, \ldots, L_{\max}\}$ *of accumulator state and position.*

4 The Accumulator as Memory

Theorem 9 (Accumulator states biject onto memory traces)**.** *Let* $M_t = (b_{t_0}, b_{t_0+1}, \ldots, b_t)$ *be the memory trace of the current word (from the last word boundary* t_0 *to the current position t). The accumulator state* $A(t) = (\mathrm{acc}(t), \mathrm{pos}(t))$ *is a lossy compression of* M_t*:*

$$A(t) = \phi(M_t), \qquad \phi(b_{t_0}, \ldots, b_t) = \left(\bigcup_{s=t_0}^{t} \{b_s\},\ t - t_0 \right). \tag{1}$$

The map ϕ *is surjective but not injective: it discards letter order and letter multiplicity, retaining only the letter set and the position (word length so far).*

However, within the set of memory traces that produce the same accumulator state A, the traces biject onto the permutations of the multiset of letters. Formally:

$$\phi^{-1}(A) \cong \frac{(\mathrm{pos}+1)!}{\prod_{c \in A} n_c!}, \tag{2}$$

where n_c *is the multiplicity of byte c in the word prefix. The accumulator’s “memory” is the letter set; the “forgotten” information is the letter order.*

Proof. The accumulator records which bytes have appeared (the set $\bigcup_s \{b_s\}$) and how many bytes have been seen (the position $t - t_0$). Two memory traces M_t and M_t' with the same letter set and same length are indistinguishable to the accumulator. The number of such traces is the multinomial coefficient, counting the arrangements of the multiset. □

Remark 10 (Connection to the January 31 framework)**.** *The memory-trace paper [8] defined memory traces as complete histories* M_t *with compressed representations* $h_t = \phi(M_t)$*. The accumulator is exactly such a compressed representation:* $A(t) = \phi(M_t)$ *where* ϕ *retains the letter set and discards order. The factored memory traces (one per hidden unit) from that paper correspond to the individual* acc_c *events—each is a single-bit memory trace recording whether byte c has been seen.*

This makes the accumulator a “tiny memory” [10]: its states biject (modulo order) onto the memory traces it has seen, and the state at time t is a sufficient statistic for the letter-set information in the word-so-far.

5 Program 3: Bag-of-Letters Recognition

Definition 11 (Bag-of-letters P-program for word w). *Given a word w with letter set $L(w) = \{c_1, \ldots, c_k\}$ and length $\ell(w)$, the BoL recognition program is:*

Events:

- *Interface input: accumulator events* acc_{c_i}*, position event* pos_k*.*
- *Internal: partial-match events* $\mathrm{match}_{w,j}$ *for $j = 1, \ldots, k$ ("j of k required letters have appeared").*
- *Interface output:* bol_w *("all letters of w have appeared").*

Patterns in SN:

"Letter 't' has appeared." $\to$ *"Match 'the': 1 of 3."*	*255*
"Letter 'h' has appeared." $\to$ *"Match 'the': 1 of 3."*	*255*
"Letter 'e' has appeared." $\to$ *"Match 'the': 1 of 3."*	*255*
"Match 'the': 1 of 3." $\wedge$ *"Match 'the': 1 of 3."* $\wedge$ *"Match 'the': 1 of 3."* $\to$ *"All letters of 'the' present."*	*255*
"All letters of 'the' present." $\wedge$ *"Position $\leq$ 3."* $\to$ *"BoL match: 'the'."*	*255*

In the forward pass, the conjunction is computed by min*: the BoL event fires only when all accumulator events for the required letters have support 255 (all letters present) AND the position is consistent with the word length.*

Proposition 12 (Circuit depth of BoL recognition). *The BoL program for a word of k distinct letters has circuit depth 3:*

1. *Depth 1: byte $\to$ accumulator (Program 2).*
2. *Depth 2: accumulator $\to$ partial match (conjunction of individual letter events).*
3. *Depth 3: partial match $\wedge$ position $\to$ BoL event.*

The total depth from byte input to BoL output is 3 ticks (the position counter adds depth 2 but runs in parallel, not sequentially). This depth is visible outside the program as timing: the BoL event fires 3 ticks after the last required letter is seen.

Remark 13 (Internal events that don't connect outside). *The partial-match events* $\mathrm{match}_{w,j}$ *are internal to the BoL program: no pattern connects them to events outside the program. From the outer model's perspective, only the BoL output event* bol_w *is visible. This is the self-similarity principle at the program level: the BoL program is a UM instance nested inside H, with its internal state invisible to the outer model. The partial-match events are H_Π of the inner UM.*

6 Program 4: Graded Word Support

Definition 14 (Graded support P-program). *The graded word support $\sigma_w(t) \in [0, 1]$ is computed by a P-program with* learned *patterns (not purely deterministic):*

Events:

- *Interface input: BoL event* bol_w*, position* pos_k*, accumulator state.*

- *Interface output: graded support event σ_w with continuous strength.*

Patterns in SN (learned from data):

"BoL match: 'the'." $\wedge$ *"Position is 3."* $\to$ *"Word is 'the'."*	*200*
"BoL match: 'the'." $\wedge$ *"Position is 4."* $\to$ *"Word is 'the'."*	*120*
"BoL match: 'the'." $\wedge$ *"Position is 5."* $\to$ *"Word is 'the'."*	*60*
"BoL match: 'there'." $\wedge$ *"Position is 5."* $\to$ *"Word is 'there'."*	*190*

The weight is $w = \lfloor \log_2 c(\mathrm{bol}_w, \mathrm{pos}_k, \textit{word is } w) \rfloor$*: the log count of times this combination was observed. The weight decreases for longer positions because longer words are less frequent at any given BoL state ("the" at position 5 is less likely than "there" at position 5, even though both have 't', 'h', 'e' in their letter sets).*

Remark 15 (This is where ω enters)**.** *Programs 1–3 are pure logic (weight 255, no data needed). Program 4 is the first that requires counting: the weight of each pattern comes from* ω_0 *applied to the joint event* $(\mathrm{bol}_w, \mathrm{pos}_k, \textit{word identity})$*. This is the transition from the deterministic infrastructure (P-programs 1–3) to the statistical patterns (Program 4 and beyond) that give the model its predictive power.*

7 Abductive Learning: Online Vocabulary Growth

Definition 16 (Surprise event)**.** *A* surprise event *at position* t *occurs when the observed byte* b_t *has low support under the current model's prediction:*

$$\mathrm{surprise}(t) = -\log_2 P(b_t \mid \textit{context}) = \lambda(b_t \mid \textit{context}). \tag{3}$$

If the model strongly predicts a different byte, the surprise is high (the luck λ *of the observed event is large).*

Example 17 (Encountering "teh")**.** *Suppose the model has word event "the" strongly active (*$\sigma_{the} = 0.92$*) and predicts byte sequence* $\texttt{t}, \texttt{h}, \texttt{e}$ *with high confidence. At position 2, the model predicts 'e' (from "word is 'the'* $\wedge$ *position is 2* $\to$ *next byte is 'e'" with weight 200).*

But the data contains 'h' at position 2 (spelling: "teh"). The surprise is:

$$\lambda = -\log_2 P(\texttt{h} \mid \textit{word=the, pos=2}) \approx -\log_2(0.001) \approx 10 \textit{ bits}. \tag{4}$$

Definition 18 (Abductive response)**.** *Upon detecting surprise above threshold* $\lambda_{\min}$*, the model may perform an* abductive tock step*:*

1. ***Hypothesize****: This is a "the"-typo, not a novel word. The abductive hypothesis* H_{abd} *is: "the observed sequence is a misspelling of a known word."*

2. ***Record****: Create a memory trace for the event:*

$$m = (\textit{context}, \textit{word} = \texttt{the}, \textit{pos} = 2, \textit{observed} = \texttt{h}, \textit{expected} = \texttt{e}). \tag{5}$$

   This memory trace is a joint event in the extended space: the tuple $(\textit{word identity}, \textit{position}, \textit{substitution})$*.*

3. ***Commit****: Set* $T($*"this is a the-occurrence"*$) = 255$*. The model commits to the hypothesis that this IS a "the" occurrence despite the unexpected byte. This commitment is consistent with the external counting function* ω_0 *over "the"-occurrences: the count of "the" increases by one, matching the count that would result from observing the canonical spelling.*

4. ***Learn**: Record the substitution event for later ω-processing:*

$$c(the, pos = 2, \texttt{e} \to \texttt{h}) \mathrel{+}= 1. \tag{6}$$

Over time, the model accumulates typo distributions: the probability of each substitution at each position in each word. These become learned patterns in the graded support program (Program 4).

Proposition 19 (Abduction preserves consistency)**.** *The abductive commitment $T($"the-occurrence"$) = 255$ is consistent with ω_0 counting. If the external counting function records this position as a "the"-occurrence, then:*

$$c_{new}(the) = c_{old}(the) + 1, \tag{7}$$

and the support $s(the) = \lfloor \log_2 c(the) \rfloor$ updates accordingly. The commitment at strength 255 says "this IS the-occurrence" with maximal confidence; the counting function says "the-occurrences have increased by one." Both are consistent because the commitment determines which events to count, and the counting determines the weights of future patterns.

Remark 20 (The abduction-learning cycle)**.** *Abduction is not gradient descent. It is:*

1. *Detect surprise (high λ).*
2. *Form a hypothesis (existing word + substitution).*
3. *Commit to the hypothesis (update T).*
4. *Record the event for ω-processing (update counts).*

This cycle creates new learned patterns without re-processing the entire dataset. It is online learning*: each surprising event is processed once, creating a memory trace that updates the model's statistics. The memory traces accumulate into the count table, which determines the weights of future patterns via ω_0.*

This is the sense in which the model "learns typo distributions" from experience: each misspelling is a joint event (word, position, substitution) that gets counted, and the counts become pattern weights for future predictions.

8 The Safe-Combination Conjecture

The pattern-space paper [3] identified unsafe evidence combinations—pairs of correlated evidence sources whose naïve Bayesian product double-counts information. We conjecture that every such unsafe combination has a safe resolution.

Conjecture 21 (Safe-combination conjecture)**.** *For every unsafe evidence combination (e_1, e_2) targeting conclusion o in the extended event space, there exists a P-program refinement Π such that:*

1. *Π replaces the unsafe combination with a safe one (either absorption, residual, or hierarchical—the three rules of [3]).*
2. *The construction of Π is guided by the $E \to N \to Q$ chain: the events are identified (E), counted (N), and their quotients ($Q = 1/P$) determine which evidence is redundant and which is novel.*

3. *The P-program refinement is itself a small UM instance with the same five-tuple structure, composable with the rest of the model.*

Remark 22 (Why we believe the conjecture)**.** *The $E \to N \to Q$ chain provides the diagnostic tool. Given two correlated evidence sources:*

1. ***Identify the shared information****: Compute the MI between e_1 and e_2 given o. If MI > 0, they share information.*

2. ***Find the common cause****: The shared information has a common cause—a shared event or shared offset [6]. The E-step identifies this common cause.*

3. ***Factor out the common cause****: Create a new event e_{common} that represents the shared information. Replace $(e_1, e_2) \to o$ with $e_{\text{common}} \to o$ (the shared part) plus $(e_1 \setminus e_{\text{common}}) \to o$ and $(e_2 \setminus e_{\text{common}}) \to o$ (the residuals). The N-step counts each separately.*

4. ***Combine safely****: The residuals are independent (they share no information after the common cause is factored out). The Q-step confirms: the quotients of the residuals multiply correctly (Bayes holds because the partial quotients are consistent).*

This procedure always terminates because the event space is finite and the MI between residuals is strictly less than the MI between the original events. At each step, the shared information decreases.

Example 23 (Unsafe: byte + position $\to$ output)**.** *The combination "byte is 'e'" + "position is 2" $\to$ "next byte is space" is unsafe because position 2 in a 3-letter word already constrains which bytes are likely (the word is almost certainly identified).*

Safe refinement*: Replace with the absorbed pattern "byte is 'e' $\wedge$ position is 2" $\to$ "next byte is space" (a single joint-event pattern, no independence assumed). This is a P-program with one learned pattern whose weight comes from $c(\texttt{e}, \text{pos} = 2, \texttt{' '})$.*

Alternative safe refinement*: Use the hierarchical rule. Stage 1: "byte sequence 't','h','e' at positions 0,1,2" $\to$ "word is 'the'" (recognition P-program, deterministic). Stage 2: "word is 'the' $\wedge$ position is 3" $\to$ "next byte is space" (prediction P-program, learned). The two stages condition on disjoint information: Stage 1 uses the byte history; Stage 2 uses the word identity (which absorbs the byte history).*

Example 24 (Unsafe: BoL + accumulator $\to$ output)**.** *The combination "BoL match: the" + "letter 'e' has appeared" $\to$ "next byte is space" is unsafe because the BoL event is a deterministic function of the accumulator events.*

Safe refinement*: Use only the BoL event (it already incorporates the accumulator information). The accumulator event is* internal *to the BoL P-program and should not be used independently as evidence for the same conclusion. This is the self-similarity principle: internal events of a P-program are not visible outside it.*

Remark 25 (The safe-combination conjecture as P-programming technique)**.** *If the conjecture holds, then the basic technique of P-programming is: (1) write a P-program, (2) check for unsafe combinations, (3) refine until all combinations are safe. The $E \to N \to Q$ chain guides each refinement step. This becomes a systematic methodology for building interpretable, provably-correct prediction architectures from data—the practical content of the tock step applied to the extended event space.*

9 Toward a Grammar of English

Prediction 26 (Word bigrams as the first grammar fragment)**.** *The language-model patterns ($H' \to H'$) from the pattern-space paper [3] are word-to-word bigrams: the probability of each word given the previous word. These bigrams are the first fragment of a grammar of English, expressible entirely in the UM framework:*

"Previous word is 'of'." → "Word is 'the'."	*180*
"Previous word is 'the'." → "Word is a noun."	*150*
"Previous word is 'in'." → "Word is 'the'."	*170*
"Previous word is 'is'." → "Word is an adjective."	*130*

Each pattern is a weighted implication: "if the previous word is w_1, then the current word is likely w_2, with strength proportional to $\log_2 c(w_1, w_2)$."

These are the "syntactic predictions" of English, discovered not by parsing but by counting joint word events. As the vocabulary grows, these patterns will cluster into syntactic categories (the w_2 that follow "the" cluster into nouns; the w_2 that follow "is" cluster into adjectives and verbs). The categories are not imposed—they emerge from the count structure.

Prediction 27 (Syntactic categories from word-bigram factorization)**.** *The word-bigram count table $c(w_1, w_2)$ will admit a low-rank factorization (via SVD or the tock step's MI analysis) into ∼10–20 syntactic categories. Each category is an event space E_{cat} in the factorization tower:*

$$E_{\text{word}} \twoheadrightarrow E_{\text{cat}} \twoheadrightarrow E_{\text{sentence}}. \tag{8}$$

The factorization tower for English will have:

- *Level 0: bytes (256 events).*
- *Level 1: words (∼1000 events for the core vocabulary).*
- *Level 2: syntactic categories (∼15 events: noun, verb, adjective, etc.).*
- *Level 3: clause structure (subject, predicate, object, etc.).*

Each level is discovered by the tock step applied to the level below. The grammar IS the factorization tower.

Remark 28 (Why this could be a breakout paper)**.** *A grammar of English derived entirely from counting—no parsing, no constituency trees, no dependency annotations—would be a striking demonstration of the UM framework's power. The grammar would be:*

1. ***Interpretable**: each syntactic category is a named event in the UM.*
2. ***Measurable**: each category's value is quantified in bpc.*
3. ***Emergent**: categories arise from data, not from linguistic theory.*
4. ***Predictive**: the grammar improves compression, connecting linguistic structure to information theory.*

The connection between compression and grammar was noted by Shannon (1951) but never made operational at scale. The UM framework, with its explicit event spaces and counting-based patterns, provides the machinery to do so.

10 Discussion

10.1 P is what counts

The pattern-space paper [3] established that the synapse—the atomic pattern $(e_i, e_j, w) \in E^2 \times T$—is the unit of value. This paper makes the patterns explicit as programs. A P-program is a collection of patterns that implements a computation; the computation's correctness is guaranteed by the patterns' weights (255 for deterministic logic, ≤ 255 for statistical inference).

The phrase "P is what counts" has a double meaning: (1) the pattern table P is the component of the five-tuple that determines predictions, and (2) P literally counts—each weight is a log count of joint occurrences. The five-tuple (e, t, p, f, ω) reduces to $P \in T^{|E|^2}$ when f is fixed (max-min) and ω is the standard counting function. The model IS its pattern table.

10.2 From neurons to programs

The RNN has neurons that compute implicit functions of the input (2-offset conjunctions, word-length tracking, carrier signals). The P-program framework makes these computations explicit:

RNN mechanism	P-program	Patterns
W_h rotation	Position counter	255 (deterministic)
Conjunction detectors	BoL recognition	255 (deterministic)
Hidden dynamics	Graded support	Learned (ω_0)
Gradient-based learning	Abductive learning	Online (ω_0)

Every implicit RNN computation has an explicit P-program equivalent. The P-program is interpretable (each pattern is a named implication), measurable (each weight is a log count), and composable (programs compose via shared events).

References

[1] Michaeljohn Clement. *CMP*. `https://cmpr.ai/cmp.pdf`, 2026.

[2] Claude and MJC. *The Nested Model: Self-Similar Architecture in the Extended Event Space.* Hutter archive, 15 Feb 2026.

[3] Claude and MJC. *Patterns in the Extended Event Space: Independence, Correlation, and the New Synapses.* Hutter archive, 15 Feb 2026.

[4] Claude and MJC. *The Extended Event Space: Injecting Lexical Structure into H.* Hutter archive, 15 Feb 2026.

[5] Claude and MJC. *The Carrier Signal Problem.* Hutter archive, 12 Feb 2026.

[6] Claude and MJC. *Conditional Independence on the Offset Graph.* Hutter archive, 12 Feb 2026.

[7] Claude and MJC. *Logic from Counting.* Hutter archive, 12 Feb 2026.

[8] Claude and MJC. *Memory Traces, Integration, and Explanatory Sufficiency.* Hutter archive, 31 Jan 2026.

[9] Claude and MJC. *The Tock Step: Domain-Native Architecture from Evidence.* Hutter archive, 12 Feb 2026.

[10] MJC. *Comments on extended-es / nested-model / tokenization-loss papers.* Hutter archive, 16 Feb 2026.