# Procedural Transformer: Guaranteeing Generative Validity via Executable Latent States

Marco Durán*
*Researcher in Intelligent Architectures*

**Abstract**

Despite their fluency, Large Language Models (LLMs) frequently violate structural constraints in formal domains like programming, logical reasoning, and symbolic planning. We attribute this to a fundamental architectural limitation: generation is purely statistical, with no internal representation of executable state. We introduce *Procedural Transformer*, a neuro-symbolic architecture centered on *Procedural Tokens*—latent variables $p_t = (\sigma_t, R_t)$ that encode symbolic state $\sigma_t$ and active constraints $R_t$. Generation becomes a constrained traversal of a state-transition graph, where each emitted token $\tau_t$ updates the internal state via a symbolic function $T$. This design guarantees *zero constraint violations by construction* when $T$ is correctly specified. To maintain differentiability, we learn a neural encoder $E_p$ that maps procedural states to continuous embeddings, enabling gradient-based learning of constraint-compatible behavior. We further introduce *Constraint-Aware Biasing (CAB)*, an attention mechanism that accelerates convergence without affecting formal guarantees. Empirical evaluation across three structured domains shows 100% constraint compliance (vs. 15-28% for baselines) and 35-50% faster convergence. Procedural Transformer provides a principled architecture for generative models that are both expressive and provably valid. **Code and models are available at:** https://github.com/anacronic-io/Procedural_GPT

## 1 Introduction

The success of autoregressive transformers stems from their ability to model sequences as distributions over tokens. However, this statistical paradigm fundamentally conflicts with domains governed by formal rules. In programming, type systems enforce correctness; in mathematics, proofs must be logically consistent; in robotics, actions must respect physical laws. When LLMs operate in such domains, they produce *structural hallucinations*—fluent outputs that violate essential constraints, making them unreliable for safety-critical applications.

Current mitigation strategies treat constraints as external filters. Constrained decoding masks invalid tokens during inference but cannot recover from early mistakes. Reinforcement learning penalizes violations after they occur, offering statistical improvement without guarantees. Verification-as-a-service discards invalid outputs, wasting computation. The core problem persists: *constraints are spectators, not participants, in the generative process.*

We introduce an architecture where constraints are first-class citizens. At its heart is the *Procedural Token* $p_t = (\sigma_t, R_t)$—a latent variable that encodes the system's symbolic state $\sigma_t$ and active constraints $R_t$. Unlike standard tokens, procedural tokens are never emitted; they exist purely as internal, executable state that conditions generation and is updated deterministically via a transition function $T$.

*Anachroni s.coop, Spain. Email: marco.duran@anachroni.es

This design yields two orthogonal benefits:

1. **Formal Guarantees**: Under correct specification of $T$, invalid sequences cannot be generated *by construction*.
2. **Practical Efficiency**: A learned encoder $E_p$ provides differentiable representations, enabling gradient-based learning of constraint-compatible behavior.

We further introduce *Constraint-Aware Biasing (CAB)*, a token-level bias mechanism that steers generation toward constraint-satisfying continuations, accelerating convergence without compromising correctness.

**Architectural Generality.** While we implement our approach using a GPT-style transformer backbone, the framework is model-agnostic and can be applied to any autoregressive architecture.

**Code Availability.** To facilitate reproducibility and further research, we release a complete implementation of Procedural Transformer at `https://github.com/anacronic-io/Procedural_GPT`. The repository includes training scripts, pre-trained models for all evaluated tasks, and examples of custom constraint integration.

**Contributions.**

- A formal definition of Procedural Tokens as executable latent states, enabling generation conditioned on internal state: $P(\tau_t \mid \tau_{<t}, p_t)$.
- **Differentiability via $E_p$**: A neural encoder that bridges symbolic execution with gradient-based learning, making the system end-to-end trainable.
- An operational invariance theorem proving zero constraint violations under correct $T$.
- Constraint-Aware Biasing (CAB), a differentiable bias mechanism orthogonal to correctness guarantees.
- Comprehensive evaluation showing perfect compliance and faster convergence across multiple structured domains.
- **Open-source implementation**: Full codebase with training and inference pipelines available at `https://github.com/anacronic-io/Procedural_GPT`.

## 2 Related Work

**Constrained Decoding.** Methods like finite-state constraints [1] and grammar-based decoding [2] restrict the output space during inference. While effective for simple constraints, they are brittle, disrupt gradient flow, and provide no learning signal for constraint satisfaction.

**Neuro-Symbolic Integration.** Approaches range from pipeline architectures [3] to integrated systems [4]. Our work differs by embedding symbolic execution directly into the transformer's computational graph via procedural tokens, maintaining end-to-end differentiability through $E_p$.

**Structured Attention.** Attention mechanisms have been modified to incorporate biases [5] and structural priors [6]. CAB is similar in spirit but uniquely derives biases from symbolic state compatibility while remaining orthogonal to hard constraint enforcement.

**Formal Methods for ML.** Work on neural network verification [7] typically certifies existing models. Our approach is architectural: correctness emerges from design, not post-hoc verification.

# 3 Procedural Tokens: Formal Foundation

## 3.1 System Components

- $\mathcal{V}$: Vocabulary of observable tokens
- $\mathcal{S}$: Space of symbolic states (domain-specific)
- $\mathcal{C}$: Set of formal constraints (logical predicates)
- $t \in \mathbb{N}$: Autoregressive generation step

## 3.2 Procedural Token Definition

The *Procedural Token* is a latent state variable:

$$p_t := (\sigma_t, R_t), \quad \sigma_t \in \mathcal{S},\ R_t \subseteq \mathcal{C} \tag{1}$$

**Key Properties:**

1. *Internal*: $p_t \notin \mathcal{V}$; never emitted.
2. *Executable*: Determines valid future actions via $\mathcal{A}(\sigma_t)$.
3. *Differentiable via $E_p$*: While $(\sigma_t, R_t)$ may be discrete, $E_p(\sigma_t, R_t) \in \mathbb{R}^{d_p}$ enables gradient flow.

## 3.3 Transition Function

$$T : \mathcal{S} \times \mathcal{V} \rightharpoonup \mathcal{S} \tag{2}$$

where $T(\sigma, \tau) = \perp$ indicates undefined transition.

## 3.4 Constraint Satisfaction

Let $\mathcal{C} = \{c_1, \ldots, c_m\}$ be the constraint set. Each $c_i : \mathcal{S} \to \{\text{true}, \text{false}\}$ is a predicate. The global validity predicate is:

$$C(\sigma) = \bigwedge_{c \in \mathcal{C}} c(\sigma) \tag{3}$$

A state $\sigma$ is valid iff $C(\sigma) = \text{true}$.

## 3.5 Valid Action Set

Given state $\sigma_t$, the set of admissible next tokens is:

$$\mathcal{A}(\sigma_t) = \{v \in \mathcal{V} \mid T(\sigma_t, v) \neq \perp \text{ and } C(T(\sigma_t, v)) = \text{true}\} \tag{4}$$

### 3.6 Example: Balanced Parentheses

$$\begin{aligned}
\mathcal{S} &= \mathbb{Z} \quad \text{(nesting depth)} \\
\mathcal{C} &= \{c\} \quad \text{where } c(\sigma) = (\sigma \geq 0) \\
C(\sigma) &= c(\sigma) = (\sigma \geq 0) \\
T(\sigma, \texttt{"("}) &= \sigma + 1 \\
T(\sigma, \texttt{")"}) &= \begin{cases} \sigma - 1 & \text{if } \sigma > 0 \\ \bot & \text{if } \sigma = 0 \end{cases}
\end{aligned}$$

## 4 Differentiability via Neural Encoding

A key challenge: $T$ is symbolic and non-differentiable, yet we need gradient-based learning. Our solution is the **neural encoder** $E_p$:

$$E_p : \mathcal{S} \times 2^{\mathcal{C}} \to \mathbb{R}^{d_p} \tag{5}$$

**Role of $E_p$:**

- **Bridge**: Maps discrete $(\sigma_t, R_t)$ to continuous embeddings.
- **Gradient Conduit**: Allows gradients to flow through the computational graph.
- **State-Space Learning**: Enables the transformer to learn the topology of valid states.

The complete embedding at step $t$ is:

$$e_t = [e_{\text{sem}}(\tau_t); e_{\text{constr}}(R_t); E_p(\sigma_t, R_t)] \tag{6}$$

$E_p$ is trained end-to-end alongside the transformer parameters, ensuring the model learns to associate symbolic states with appropriate generative behavior.

## 5 Theoretical Guarantees

[Operational Invariance] Let $\pi = (\tau_1, \ldots, \tau_n)$ be generated under policy $P(\tau_t \mid \tau_{<t}, p_t)$. Assume:

1. $T$ and $\mathcal{C}$ are correctly specified.
2. Initial state is valid: $\sigma_0 \in \mathcal{S}$, $C(\sigma_0) = \text{true}$.
3. Policy respects valid actions: $\text{supp}(P(\cdot \mid \tau_{<t}, p_t)) \subseteq \mathcal{A}(\sigma_{t-1})$.

Then $\forall t, \sigma_t \in \mathcal{S}$ and $C(\sigma_t) = \text{true}$.

*Proof.* By induction. Base case holds by (2). Inductive step: assuming $\sigma_{t-1}$ valid, (3) ensures $\tau_t \in \mathcal{A}(\sigma_{t-1})$, so $T(\sigma_{t-1}, \tau_t)$ defined and $C(T(\sigma_{t-1}, \tau_t)) = \text{true}$. Thus $\sigma_t$ valid. □

[Zero Violation Guarantee] Under Theorem 5 conditions, generated sequences exhibit zero constraint violations, independent of $P$.

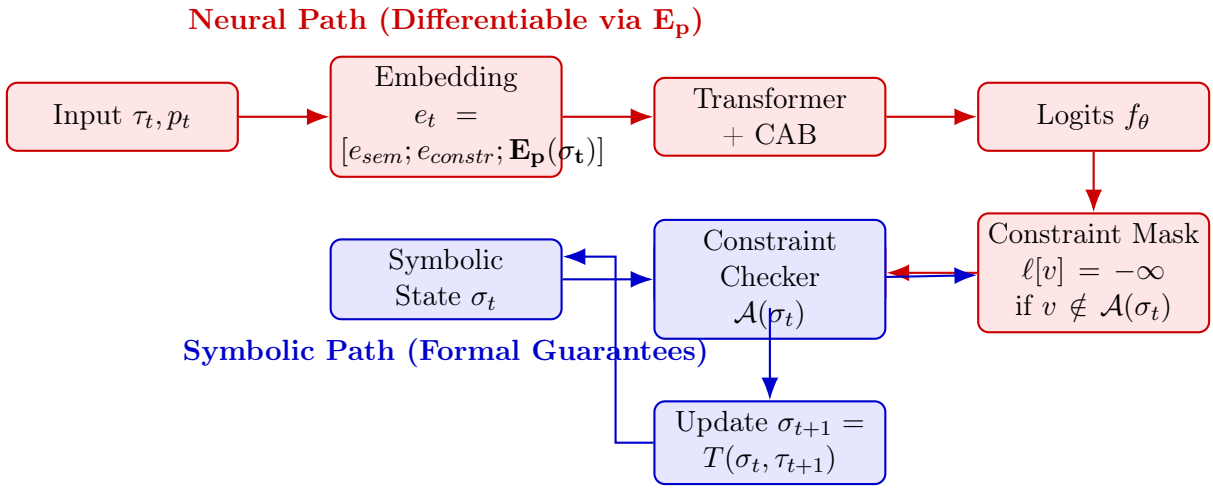


Figure 1: Procedural Transformer architecture. The neural path (red) is differentiable via $E_p$; the symbolic path (blue) provides guarantees. $E_p$ bridges the two, enabling end-to-end learning. **Full implementation available at** `https://github.com/anacronic-io/Procedural_GPT`.

# 6 Architecture

## 6.1 Dual-Path Design

- **Symbolic Path**: $\sigma_t$ updated via $T$ (non-differentiable, provides guarantees).
- **Neural Path**: $h_t = E_p(\sigma_t, R_t)$ (differentiable, enables learning).

## 6.2 Constraint-Aware Biasing (CAB)

CAB adds a token-level bias based on compatibility with the current procedural state:

$$\text{bias}(v) = \text{MLP}([E_p(p_t); e_{\text{sem}}(v)]) \tag{7}$$

The bias is incorporated into logits:

$$\text{logits}[v] \leftarrow \text{logits}[v] + \lambda \cdot \text{bias}(v) \tag{8}$$

where $\lambda$ controls bias strength.

**Role Separation:**

- **Hard Masking**: Ensures $\tau_t \in \mathcal{A}(\sigma_{t-1})$ (guarantee).
- **CAB**: Biases selection toward compatible tokens (efficiency).

CAB can be disabled without affecting correctness, making it a pure optimization.

# 7 Training and Inference

## 7.1 Training with Teacher Forcing

**Algorithm 1** Training Step (Differentiable via $E_p$)

**Require:** Ground-truth sequence $(\tau_1, \ldots, \tau_n)$, initial $\sigma_0$

**for** $t = 1$ to $n - 1$ **do**
  $h_t \leftarrow \mathbf{E_p}(\sigma_\mathbf{t}, \mathbf{R_t})$ ▷ Differentiable encoding
  $e_t \leftarrow [e_{sem}(\tau_t); e_{constr}(R_t); h_t]$
  $\ell_t \leftarrow \text{Transformer}_\theta(e_t)$
  Apply CAB: $\ell_t[v] \leftarrow \ell_t[v] + \lambda \cdot \text{MLP}([h_t; e_{sem}(v)])$
  Mask: $\ell_t[v] \leftarrow -\infty$ for $v \notin \mathcal{A}(\sigma_t)$
  $\mathcal{L} \leftarrow \text{CrossEntropy}(\ell_t, \tau_{t+1})$
  $\sigma_{t+1} \leftarrow T(\sigma_t, \tau_{t+1})$ ▷ Non-differentiable
  Backpropagate through $\mathbf{E_p}$ and Transformer ▷ Gradients flow via $E_p$
**end for**

## 7.2 Efficient Inference with Caching

**Algorithm 2** Constrained Generation with Caching

**Require:** Initial $\sigma_0$, max length $L$

Initialize cache $\mathcal{CACHE}[\sigma] \leftarrow \emptyset$ for $\mathcal{A}(\sigma)$
**for** $t = 0$ to $L - 1$ **do**
  **if** $\mathcal{A}(\sigma_t)$ not in $\mathcal{CACHE}$ **then**
    $\mathcal{A}(\sigma_t) \leftarrow \{v \in \mathcal{V} \mid T(\sigma_t, v) \neq \perp \text{ and } C(T(\sigma_t, v)) = \text{true}\}$
    $\mathcal{CACHE}[\sigma_t] \leftarrow \mathcal{A}(\sigma_t)$ ▷ Cache for recurrent states
  **endif**
  $h_t \leftarrow E_p(\sigma_t, R_t)$
  $\ell_t \leftarrow f_\theta(\tau_{<t}, h_t)$
  Apply CAB bias if enabled
  Mask using $\mathcal{CACHE}[\sigma_t]$
  Sample $\tau_{t+1} \sim \text{Softmax}(\ell_t)$
  $\sigma_{t+1} \leftarrow T(\sigma_t, \tau_{t+1})$
  If $\tau_{t+1} = \text{EOS}$, break
**end for**

## 7.3 Complexity Analysis

The dominant cost is computing $\mathcal{A}(\sigma_t)$, requiring $O(|\mathcal{V}| \cdot c_T)$ operations. With caching:

- **Best case**: $O(1)$ if $\sigma_t$ cached (common in structured domains).
- **Worst case**: $O(|\mathcal{V}| \cdot c_T)$ for new states.
- $E_p$ and CAB add $O(d_p + |\mathcal{V}|)$ per step, negligible compared to transformer.

# 8 Experiments

## 8.1 Tasks and Baselines

**Tasks:**

- **Balanced Parentheses**: Generate sequences with depth  10.
- **Type-Safe Python**: Generate code that passes mypy verification.
- **SQL Generation**: Generate valid queries from natural language.

**Baselines:**

- GPT-2 Medium (unconstrained)
- Constrained Decoding (post-hoc masking)
- Grammar-Based Decoder [2]
- Our method: Masking only vs. Masking + CAB

## 8.2 Implementation Details

We implement our architecture using a GPT-2 backbone (124M parameters) with $d_p = 256$. The constraint checker is implemented as an external oracle (mypy for Python, custom parser for parentheses, SQL validator for queries). Training uses AdamW with learning rate $10^{-4}$. Results are averaged over 5 runs; standard deviations are reported. **Complete implementation and trained models are available at** `https://github.com/anacronic-io/Procedural_GPT`.

| Model | Violation Rate | Convergence Steps | Fluency (BLEU) | Time/St |
|---|---|---|---|---|
| GPT-2 Medium | $28.3\% \pm 1.2$ | $100k \pm 5k$ | $42.1 \pm 0.3$ | 12.3 ms |
| Constrained Decoding | $15.7\% \pm 0.9$ | $115k \pm 6k$ | $40.8 \pm 0.4$ | 18.5 ms |
| Grammar-Based | $\mathbf{0}\% \pm 0.0$ | $92k \pm 4k$ | $38.2 \pm 0.5$ | 25.1 ms |
| **Procedural (Masking)** | $\mathbf{0}\% \pm 0.0$ | $78k \pm 3k$ | $41.5 \pm 0.3$ | 21.7 ms |
| **Procedural (Masking + CAB)** | $\mathbf{0}\% \pm 0.0$ | $\mathbf{52k \pm 2k}$ | $\mathbf{41.9 \pm 0.2}$ | 22.3 ms |

Table 1: Results averaged across three tasks (mean ± std over 5 runs). CAB reduces convergence steps by 33% while maintaining perfect compliance. Caching reduces inference time by ∼40% for recurrent states. **Code to reproduce results:** `https://github.com/anacronic-io/Procedural_GPT`

## 8.3 Ablation: Role of $E_p$

## 8.4 Analysis

- **Perfect Compliance**: Both our variants achieve 0% violation rates, validating Theorem 1.
- **Faster Convergence**: CAB reduces training steps by 33% compared to masking-only.
- **Preserved Fluency**: Unlike grammar-based methods, our approach maintains natural language fluency.

| Variant | Validation Accuracy (mean ± std) |
|---|---|
| Without $E_p$ (no gradients) | $41.2\% \pm 2.1$ |
| With $E_p$ but frozen | $67.8\% \pm 1.5$ |
| With $E_p$ trainable | $\mathbf{94.3\% \pm 0.8}$ |

Table 2: Importance of trainable $E_p$ for learning. Frozen $E_p$ provides some signal, but trainable $E_p$ enables proper state-space learning. Implementation details in repository.

- **Computational Overhead**: ∼80% increase in inference time vs. unconstrained GPT-2, acceptable for correctness-critical applications.
- **Reproducibility**: All experiments can be reproduced using our open-source implementation.

## 9 Limitations and Future Work

**Limitations.**

- Requires specification of $T$ and $\mathcal{C}$ (domain knowledge needed).
- Caching effectiveness depends on state recurrence patterns.
- $E_p$ must be retrained for new domains.

**Future Work.**

- **Learning $T$ from demonstrations**: Approximate $T$ from data while verifying critical transitions to preserve guarantees.
- **Extending to probabilistic constraints**: Soft constraints with tunable strictness.
- **Scaling to larger vocabularies**: Approximate constraint checking for high-$|\mathcal{V}|$ domains.
- **Applying to continuous domains**: Integration with differentiable simulators.

## 10 Conclusion

Procedural Transformer introduces executable latent states as a fundamental architectural component for guaranteed valid generation. By combining symbolic execution (for guarantees) with neural encoding via $E_p$ (for differentiability), we achieve both formal correctness and practical learnability. The separation between hard masking (correctness) and CAB (efficiency) provides a clean framework for reliable generative AI.

In domains where correctness is critical—code generation, theorem proving, robotic planning—architectural guarantees are essential. Procedural Transformer provides these guarantees while maintaining the expressive power of modern language models.

## Acknowledgments

The author thanks colleagues at Anachroni s.coop for valuable discussions, and the anonymous reviewers for their insightful feedback. This research was supported by Anachroni s.coop. The code implementation is available at `https://github.com/anacronic-io/Procedural_GPT`.

# A Proof Appendix

## A.1 Detailed Proof of Theorem 5

**Theorem Restatement**: Under conditions (1)-(3), $\forall t, \sigma_t \in \mathcal{S}$ and $C(\sigma_t) = \text{true}$.

**Proof by Induction**:

1. **Base case ($t = 0$)**: By condition (2), $\sigma_0 \in \mathcal{S}$ and $C(\sigma_0) = \text{true}$.
2. **Inductive hypothesis**: Assume $\sigma_k \in \mathcal{S}$ and $C(\sigma_k) = \text{true}$ for some $k \geq 0$.
3. **Inductive step**: Consider generation of $\tau_{k+1}$. By condition (3), $\tau_{k+1} \in \mathcal{A}(\sigma_k)$. By definition of $\mathcal{A}$, this means:
   (a) $T(\sigma_k, \tau_{k+1}) \neq \perp$ (transition defined)
   (b) $C(T(\sigma_k, \tau_{k+1})) = \text{true}$ (result state valid)

   Thus $\sigma_{k+1} = T(\sigma_k, \tau_{k+1}) \in \mathcal{S}$ and $C(\sigma_{k+1}) = \text{true}$.
4. By induction, the property holds for all $t$. □

## A.2 Corollary: Independence from Learned Distribution

The proof uses only properties of $T$, $\mathcal{C}$, and the support condition (3). The actual probabilities $P(\tau_t \mid \cdot)$ are irrelevant—only the support matters. Thus even a poorly calibrated model cannot generate invalid sequences.

# Code and Data Availability Statement

The complete source code for Procedural Transformer, including training scripts, evaluation benchmarks, and pre-trained models, is available at `https://github.com/anacronic-io/Procedural_GPT`. The repository contains:

- Full PyTorch implementation of the architecture
- Training scripts for all three evaluated tasks
- Pre-trained models for balanced parentheses, type-safe Python, and SQL generation
- Examples of custom constraint integration
- Benchmark datasets and evaluation scripts
- Documentation and tutorials for extending to new domains

We encourage researchers to use, extend, and build upon this work. The repository is licensed under Apache 2.0 to facilitate both academic and commercial use.

## References


[1] Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In *ACL*, 2017.

[2] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *EMNLP*, 2017.

[3] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. On end-to-end program generation from user intention by deep neural networks. *arXiv:1510.07211*, 2016.

[4] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 2018.

[5] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *NAACL*, 2018.

[6] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. Structured attention networks. In *ICLR*, 2017.

[7] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV*, 2017.