

NEAT: Semantic Prefix Analysis for Sound Constrained Code Generation with Large Language Models

Paul Kronlund-Drouault

École Normale Supérieure de Lyon, Lyon, France
`paul.kronlund-drouault@ens-lyon.fr`

Abstract. Large language models are increasingly used as probabilistic search engines for code generation, theorem proving, and structured tool use. Existing constrained decoding systems provide strong guarantees for regular or context-free output formats, but they do not directly account for context-dependent semantic constraints such as scope, type compatibility, declaration effects, or language-specific well-formedness predicates. We introduce semantic prefix grammars (SPG), a declarative framework that reduces context-aware constrained generation to an online static analysis problem. An SPG pairs context-free productions with binding annotations and semantic premises over a decidable, prefix-conservative abstract domain. A token is admissible exactly when the combined syntactic and semantic state remains extendable to at least one complete semantically valid expression. This yields a formal soundness invariant: every oracle-accepted prefix has a valid completion. We implement SPG in Aufbau, a Rust prefix parser based on Earley-style saturation with typed obligation stores and right-frontier context flow, and expose the oracle through P7, a constrained generation layer for Hugging Face Transformers. Across eight model families from 124M to 27B parameters, constrained mixed-mode generation improves global pass rate from 23% to 42%, with a 27B model under P7 surpassing GPT-5.4-mini on the same evaluation. These results suggest that semantic constraints can substantially improve the reliability of code generation, especially when deploying smaller open-weight models.

Keywords: Static Analysis · Constrained Decoding · Large Language Models · Type Checking · Program Synthesis

1 Introduction

Problem. Large language models are effective samplers for formal artifacts, but unconstrained sampling gives no structural guarantee: a model can always emit a token that leaves the target language, violates a type rule, or invalidates a proof script. Training, retrieval, and prompting reduce the probability of such failures, but they do not remove the failure states from the decoding process. For formal methods workloads this distinction matters: a nearly-correct decoder is still an untrusted component.

Constrained decoding addresses this problem by masking or rejecting inadmissible continuations. The best-developed line of work targets regular languages, JSON schemas, and context-free grammars. This is necessary but not sufficient for programming languages and proof-oriented languages. Many important constraints are semantic in the ordinary programming-language sense: identifiers must be in scope, declarations modify future contexts, branches must agree on types, and expression forms may be valid only under language-specific side conditions. These constraints are context-dependent even when the surface syntax is context-free.

Main idea. Building upon grammar-constrained decoding systems such as Outlines [18], subword-aligned decoding [1,10], Earley-optimized structured decoding [15], and recent type-constrained generation [7], we propose a declarative framework for prefix decoding with semantic constraints. A specification contains ordinary productions with binding annotations, plus inference-style semantic rules. The parser maintains a partial forest and a finite semantic state; a continuation is admissible exactly when parsing and semantic evaluation do not reach stable contradiction.

We reuse the standard view of attribute grammars [5] and constraint/logic grammars [11,14]: a context-free derivation is accepted only when attached semantic information admits a satisfying valuation. We name our class *semantic prefix grammars* and add the condition that the constraint domain must be decidable and its partial evaluator must be conservative over prefixes. The current instantiation supports typed languages with finite contexts, rule-local unification, binding evidence, and context transforms. Keeping the same model, richer fragments of languages such as TypeScript or Python can be obtained by extending the premise language and context structures while preserving the same parser architecture.

Contributions. This paper makes the following contributions:

1. A formal model of *prefix closure* parsing for our grammar class along with proof targets for generalized prefix conservative semantic constraints.
2. A working implementation: a Rust completion engine (AUFBAU) and a Python sampling layer (P7) that integrates with Hugging Face Transformers [19].
3. A compact prefix parsing algorithm based on Earley-style saturation [3], enriched with semantic obligations, context flow, and an end-of-input closure phase..
4. Empirical evaluation: constrained generation on zero-shot code generation with small models, and on mixed Chain-of-Thought [17] plus formal block reasoning tasks.

2 Semantic Prefix Grammars

2.1 Completability

Let $L \subseteq \Sigma^*$ be a formal language over a character alphabet Σ . A prefix $s \in \Sigma^*$ is **completable** for L when

$$\mathcal{C}_L(s) \triangleq \exists s' \in \Sigma^*, ss' \in L.$$

2.2 Grammar

A Semantic Prefix Grammar is a tuple

$$G = (N, T, P, S, \Theta, \mathcal{T}, \mathcal{B}, A)$$

where:

- N is a finite set of non-terminals
- T is a finite set of terminal recognizers over segment text
- P is a finite set of productions
- $S \in N$ is the distinguished start symbol
- Θ is a finite set of semantic rules
- $\mathcal{T} : N \rightarrow \Theta$ maps semantic non-terminals to their rule

- $\mathcal{B}$ the set of binding identifier
- $A : N \rightarrow P^*$ maps each non-terminal n to its ordered list of productions

$$A(n) = (p_1^n, \dots, p_{k_n}^n).$$

Assumption 1. *An SPG has no structural productivity constraints. We will consider the improductive case trivial in terms of syntactic and semantic constraints, and from now on assume we are dealing with only productive grammars in the SPG class.*

Productions and Alternatives

Definition 1 (Production). *A production $p_a^n \in A(n)$ is a finite sequence*

$$p_a^n = \alpha_1^a[b_1^a] \alpha_2^a[b_2^a] \cdots \alpha_m^a[b_m^a],$$

where:

- $a \in \{1, \dots, k_n\}$ is the index of the alternative;
- each $\alpha_i^a \in T \cup N$ is either a terminal recognizer or a non-terminal;
- each $b_i^a \in \mathcal{B} \cup \{\varepsilon\}$ is either a binding identifier or the empty binding.

The production p_a^n is attached to the unique non-terminal n . We write $|p_a^n| = m$ for its length.

Bindings are syntactic annotations on child positions. They don't affect the context-free structure but they name child evidence that may be referenced by semantic rules.

Segmentation. Parsing is parameterized by a segmentation policy

$$\text{seg} : \Sigma^* \rightarrow \text{Seg}^*.$$

We require **seg** to be *deterministic and prefix-aware*: an unfinished final segment may be marked open, but the segmentation of already closed interior segments is stable under extension. The segmentation algorithm is a greedy left-to-right longest-match policy. Delimiters are chosen depending on the task, but independently of the grammar.

Terminal matching is abstracted through a segment-level interface

$$\text{match}(t, x) \rightarrow \{\perp\} \cup \{\text{Prefix}, \text{Exact}, \text{Extensible}\} \times \{\Sigma^*, \varepsilon\}$$

This interface is justified by RegEx derivatives [2,9] for regular terminals. The parser itself only depends on the evidence exposed by **match**. The value $\perp$ means that the segment cannot match the terminal.

$$\text{Term}(x, \xi, r), \quad \xi \in \{\text{Prefix}, \text{Exact}, \text{Extensible}\} \quad r \in \Sigma^*$$

Terminal evidence is treated uniformly with child evidence: it may satisfy bindings, may remain open at the right frontier, and may be refined by later input. The interface **Term**(x, ξ) has at least one extension $r \in \Sigma^*$, corresponding to the derivative of x with respects to tt , such that the refined evidence for xr is **Exact**.

Binding Signatures Semantic rules refer to children of a production through binding identifiers. We restrict bindings so that every semantic non-terminal has a fixed binding interface shared by all of its alternatives.

Definition 2 (Binding signature).

The binding signature for $n \in N$ is the set $\mathcal{B}(n) \subseteq \mathcal{B}$. A grammar is binding-regular if for every $b \in \mathcal{B}(n)$, b appears only in productions of n and exactly once in every production.

$$\forall n \in N, \left(\forall \text{bin} \mathcal{B}(n), \forall a \in |A(n)| \exists! i \in |p_a^n|, b_i^a = b \right) \wedge \left(\forall n' \in N \setminus \{n\}, \forall a' \in |A(n')|, \nexists j \in |p_{a'}^{n'}|, b_j^{a'} = b \right)$$

Definition 3 (Binding position). Let G be binding-regular. For $n \in N$, $p_a^n \in A(n)$, and $b \in \mathcal{B}(n)$, define

$$\text{pos}_{n,a}(b) = \text{the unique } i \in \{1, \dots, |p_a^n|\} \text{ such that } b_i^a = b.$$

This function is well-defined by totality and uniqueness of binding signatures.

The parser uses $\text{pos}_{n,a}$ to route semantic obligations to the corresponding child. If an item is currently positioned before child i of alternative p_a^n , then the only binding obligations projected to that child are those for bindings b satisfying

$$\text{pos}_{n,a}(b) = i.$$

Property 1. By definition of pos , and using *binding regularity*, we can establish binding positions are unique :

$$b_1 \neq b_2 \iff \text{pos}_{n,a}(b_1) \neq \text{pos}_{n,a}(b_2)$$

When referring to an SPG in the case of a proof or a definition, we assume binding regularity.

2.3 Semantic graphs

The grammar productions of §2.2 expose *evidence nodes*: terminal evidence matched against the input, span evidence recognized for non-terminals, and binding references exported between siblings. Rather than treating types or contexts as external annotations, the semantic layer records semantic information as a finite *relational graph* over this evidence.

Let Σ be the terminal alphabet. For an input prefix $s \in \Sigma^*$, we write $\sigma(s)$ for the parser state reached after consuming s , and $\mathcal{G}(s)$ for the semantic graph associated with that state. We similarly write $\mathcal{E}(s)$ and $\mathcal{C}(s)$ for its evidence nodes and constraint instances.

The graph is constructed monotonically along input extension. If $r \in \Sigma^*$ and the parser can reach $\sigma(s \cdot r)$ from $\sigma(s)$, then

$$\mathcal{E}(s) \subseteq \mathcal{E}(s \cdot r) \quad \text{and} \quad \mathcal{C}(s) \subseteq \mathcal{C}(s \cdot r).$$

Thus extending the input may expose new evidence nodes and emit new constraint instances, but it does not rewrite the interior evidence already exposed by the parser.

The semantic relations themselves do not need to be tree-shaped or to terminate in designated constants. Rules may relate evidence nodes directly, for example by equality, lookup, unification, subtyping, or more complex decidable predicates.

Definition 4 (Evidence graph). Let $s \in \Sigma^*$ be an input prefix such that $\sigma(s)$ is reachable. The evidence graph at s is a finite hypergraph

$$\mathcal{G}(s) = (\mathcal{E}(s), \mathcal{C}(s))$$

where:

- $\mathcal{E}(s)$ is the finite set of evidence nodes accumulated by the parser: terminal nodes, span nodes, and binding-reference nodes;
- $\mathcal{C}(s)$ is the finite set of constraint instances emitted so far, drawn from a decidable constraint language and interpreted over $\mathcal{E}(s)$.

A constraint instance may refer both to evidence nodes and to ground symbols of the semantic domain, such as constant type evidence appearing directly in the premises.

A graph is *open* when some evidence still lies on the parser frontier: for example a terminal, span, type annotation, or binding reference has been introduced but is not yet fully realized by the consumed input. A graph is *closed* when all evidence required by its constraint instances has been realized. This semantic notion of closure for evidence graphs is distinct from language-level acceptance.

Example 1. In a typed lambda grammar, the prefix

$$\lambda x : \text{Int}$$

may already expose evidence for the binder x and for its type annotation, but the type evidence is still open: the input may continue to complete Int into Int . After reading a complete annotation and body, such as

$$\lambda x : \text{Int}. x,$$

the corresponding evidence graph may be closed.

Definition 5 (Graph grounding). Let $\mathcal{G}(s) = (\mathcal{E}(s), \mathcal{C}(s))$ be the evidence graph at a reachable input prefix s . A grounding of $\mathcal{G}(s)$ is a pair

$$\theta = (\mathcal{H}, h)$$

where $\mathcal{H}$ is a closed evidence graph in the semantic domain and

$$h : \mathcal{E}(s) \rightarrow \mathcal{E}_{\mathcal{H}}$$

maps current evidence nodes to resolved evidence nodes of $\mathcal{H}$, such that every constraint instance in $\mathcal{C}(s)$ is satisfied after applying h .

Equivalently, h is a constraint-preserving graph morphism from the current partial evidence graph into a closed semantic graph.

Definition 6 (Denotation). The denotation of $\mathcal{G}(s)$ is the set

$$\mathcal{D}(\mathcal{G}(s)) = \{ (\mathcal{H}, h) \mid h : \mathcal{G}(s) \rightarrow \mathcal{H} \text{ is a grounding into a closed evidence graph} \}.$$

It is the set of closed graph resolutions of the current evidence graph.

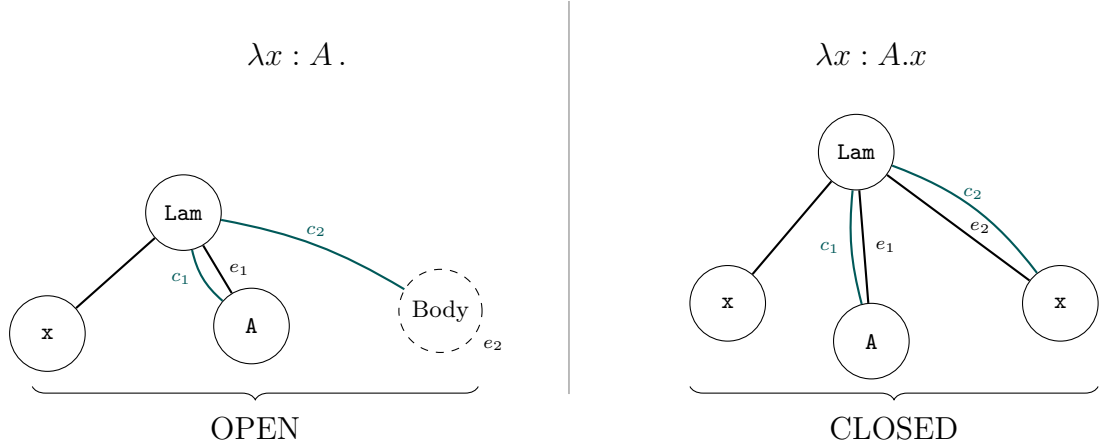


Fig. 1. Open and closed semantic graphs for prefix lambda expressions. Solid black edges are AST edges; colored curved edges are semantic links.

Definition 7 (Constraint domain). A constraint domain is a triple

$$D = (\text{Rules}, \text{Closed}, \text{eval})$$

where:

- *Rules* is a decidable language of relational rules over evidence nodes and domain constants;
- *Closed* is the class of closed evidence graphs admitted by the domain;
- *eval* is the ideal evaluator,

$$\text{eval}(\mathcal{G}(s)) \in \{\text{Satisfied}, \text{Live}, \text{Lost}\}.$$

It is defined by the following cases:

$$\text{eval}(\mathcal{G}(s)) = \text{Satisfied} \iff \mathcal{G}(s) \in \text{Closed}$$

$$\text{eval}(\mathcal{G}(s)) = \text{Live} \iff \mathcal{G}(s) \notin \text{Closed} \wedge \exists r \in \Sigma^*. \mathcal{G}(s \cdot r) \in \text{Closed}$$

The evaluator returns

$$\text{eval}(\mathcal{G}(s)) = \text{Lost}$$

otherwise.

Lemma 1 (Safe pruning). If the parser discards an input prefix s only when

$$\text{eval}(\mathcal{G}(s)) = \text{Lost},$$

then it never discards a prefix that has a closed satisfying continuation.

Proof. Suppose the parser discards s . By assumption,

$$\text{eval}(\mathcal{G}(s)) = \text{Lost}.$$

By Definition 7, this means that $D(\mathcal{G}(s)) = \emptyset$, and thus it has no map providing a continuation into a satisfiable grounding. Therefore s has no closed satisfying semantic continuation. Hence discarding s cannot remove a prefix that could still be completed into a closed satisfying graph.

We write $\text{SPG}(D)$ for grammars over a constraint domain D equipped with the evaluator of Definition 7. Grammars in this class are said to be realizable, as their semantic constraints match their syntactic realization possibilities.

3 Simple Typing Domain Implementation

This section instantiates the constraint domain of Definition 7 with a type-directed semantic algebra. The key property of this instantiation is that the ideal evaluator of Definition 7 is *computable*: the obligation store records exactly the constraints that future input must satisfy as a decidable constraint-satisfaction problem over a finite closed-type domain. The evaluator therefore does not approximate the ideal — it computes it exactly.

Closed Types and Type Evidence

A *closed type* is a fully resolved type:

$$\tau ::= t \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \vee \tau_2 \mid \neg \tau \mid \top \mid \perp.$$

Closed-type equality and inclusion are assumed decidable, with $\perp \subseteq \tau \subseteq \top$ for all τ . A *typing context* is a finite partial map $\Gamma : \text{Name} \rightarrow \text{Type}$.

Type information flows through the parser in three stages:

$$\text{type expression} \xrightarrow{\text{elab}} \text{type evidence} \xrightarrow{\text{close}} \text{closed type}.$$

Type evidence is the semantic state of a type-bearing evidence node:

$$E ::= \text{Exact}(\tau) \mid \text{Partial}(P) \mid \text{Meta}(A),$$

where $P \neq \emptyset$ is a finite set of closed-type completions, and $\text{Meta}(A)$ is a rule-local unification variable. *Type expressions* $\hat{\tau}$ in typing rules are elaborated to evidence:

$$\begin{aligned} \tau &\mapsto \text{Exact}(\tau), \\ \Gamma(x) &\mapsto \text{the evidence stored for } x \text{ in } \Gamma, \\ b &\mapsto \text{the evidence stored for binding } b, \\ ?A &\mapsto \text{Meta}(A). \end{aligned}$$

Meta-variables are rule-local and do not escape the rule application. Repeated occurrences of $?A$ within a rule are compiled into equality constraints between the corresponding nodes in the evidence graph.

Obligations

An *obligation store* Ω records the type evidence expected from bound children. Each obligation has the form

$$\omega_b = (b, v_b, E_b),$$

where b is the binding name, v_b is terminal or span evidence, and E_b is type evidence. The operator $\text{typeof}(b)$ is not primitive: it is a read of E_b from the obligation for b . Resolving a terminal child stores terminal evidence; resolving a non-terminal child stores span evidence and exported type evidence.

Lemma 2 (Obligation locality). *For a binding-regular grammar, the obligations projected to position i of alternative p_a^n are exactly*

$$\text{step}(n, a, i, \Omega) = \{ \omega_b \in \Omega \mid \text{pos}_{n,a}(b) = i \}.$$

Resolving position i may modify only these obligations.

Proof. By binding uniqueness (Property 1), each binding b has a unique position $\text{pos}_{n,a}(b)$ in p_a^n . Position i can supply evidence only for bindings whose unique position is i .

Primitive Premises

The constraint instances of Definition 4 are drawn from the following decidable premise language:

$x \in \Gamma$	name x is present in the context,
$\hat{\tau}_1 = \hat{\tau}_2$	type equality,
$\hat{\tau}_1 \subseteq \hat{\tau}_2$	type inclusion,
$\Gamma \vdash b : \hat{\tau}$	child b has type $\hat{\tau}$,
$\Gamma[x : \hat{\tau}] \vdash b : \hat{\sigma}$	child b is checked under an extended context.

Each premise is evaluated by a computable implementation of *eval* (Definition 7). We write this implementation *eval_{impl}*; Theorem 1 establishes that it coincides with the ideal evaluator on every reachable state.

Prefix-Conservativity

Lemma 3 (Type evidence is monotone). *Let e be an evidence node and let $\text{comp}(e, s)$ denote the set of closed types that the type evidence of e at state $\sigma(s)$ can resolve to. For any extension $s \cdot r$,*

$$\text{comp}(e, s \cdot r) \subseteq \text{comp}(e, s).$$

Proof. By case analysis on the evidence form.

- **Exact**(τ): $\text{comp} = \{\tau\}$, unchanged by any extension since exact evidence carries no open frontier.
- **Partial**(P): P is the finite set of closed types reachable as completions of the current type prefix. Input extension applies the regex derivative to the ongoing type segment and derivatives can only remove completions [9], so $\text{comp}(e, s \cdot r) \subseteq P$.
- **Meta**(A): before binding, comp is the full closed-type domain. Binding occurs when the obligation for A is resolved by a completed child, transitioning the evidence to **Exact** or **Partial**; thereafter the cases above apply. Binding never widens the completion set.

In all cases, $\text{comp}(e, s \cdot r) \subseteq \text{comp}(e, s)$.

Remark 1. A contra judgment is *stable*: once no completion can satisfy a premise, no further input can reverse this. This follows directly from Lemma 3 — the completion sets only shrink.

Lemma 4 (Context membership is prefix-conservative). *The premise $x \in \Gamma$ is prefix-conservative.*

Proof. At any parser item, Γ is the context at the dot, determined by the right-bound transforms of already-closed left siblings. It does not change as the input extends the current frontier item: Γ only advances when the dot moves, which is a parser step, not an input extension. Hence Γ is fixed for the duration of this analysis, and we case-split on the name evidence of x .

- **Exact**($name$), $name \in \text{dom}(\Gamma)$: the premise is **Satisfied**. Exact evidence is stable (Lemma 3), so this holds for all extensions.
- **Exact**($name$), $name \notin \text{dom}(\Gamma)$: the implementation returns **Lost**. Since Exact evidence does not change and Γ does not grow, no extension can introduce $name$ into $\text{dom}(\Gamma)$. The judgment is correctly **Lost**.
- **Partial**(P), $P \cap \text{dom}(\Gamma) \neq \emptyset$: some name completion is in scope; the premise is **Live**. **Lost** is not returned.
- **Partial**(P), $P \cap \text{dom}(\Gamma) = \emptyset$: the implementation returns **Lost**. By Lemma 3, P only shrinks under extension; since P and $\text{dom}(\Gamma)$ are already disjoint, they remain disjoint for all extensions. The judgment is correctly **Lost**.

In every case where **Lost** is returned, no grounding of any extension satisfies $x \in \Gamma$.

Lemma 5 (Type relations are prefix-conservative). *The premises $\hat{\tau}_1 = \hat{\tau}_2$ and $\hat{\tau}_1 \subseteq \hat{\tau}_2$ are prefix-conservative.*

Proof. Elaborate both sides to type evidence E_1 and E_2 . We treat equality; inclusion is symmetric. The implementation returns **Lost** only in the following cases, each of which is stable under extension by Lemma 3.

- **Exact**(τ_1) vs **Exact**(τ_2), $\tau_1 \neq \tau_2$: closed-type inequality is decidable and neither side can change. **Lost** is correct.
- **Exact**(τ) vs **Partial**(P), $\tau \notin P$: the only possible completion of E_2 is some $\tau' \in P$, and $\tau \notin P$. By Lemma 3, P only shrinks, so τ cannot enter P by extension. **Lost** is correct.
- **Partial**(P_1) vs **Partial**(P_2), $P_1 \cap P_2 = \emptyset$: no pair of completions can agree. Both sets only shrink, so the intersection remains empty. **Lost** is correct.

In all remaining cases at least one side contains a **Meta** node or the completion sets have non-empty intersection; the implementation returns **Live** or **Satisfied**, and **Lost** is never returned. Prefix-conservativity holds.

Lemma 6 (Child type ascription is prefix-conservative). *The premise $\Gamma \vdash b : \hat{\mu}$ is prefix-conservative.*

Proof. The premise reads the type evidence E_b from the obligation ω_b and elaborates $\hat{\mu}$ to evidence E_μ . It then checks $E_b = E_\mu$ or $E_b \subseteq E_\mu$ depending on the rule. Both E_b and E_μ are type evidence nodes; by Lemma 3 both are monotone under extension. The check reduces to a type relation test, so prefix-conservativity follows directly from Lemma 5.

Lemma 7 (Child-bound context ascription is prefix-conservative). *The premise $\Gamma[x : \hat{\tau}] \vdash b : \hat{\sigma}$ is prefix-conservative.*

Proof. The premise constructs the extended context $\Gamma' = \Gamma[x : E_\tau]$ where E_τ is the elaboration of $\hat{\tau}$, then checks $b : \hat{\sigma}$ under Γ' . The extension is a finite map update; it does not affect the monotonicity of E_τ under input extension (Lemma 3). The resulting check on b under Γ' is an instance of Lemma 6, so prefix-conservativity follows.

The case $E_\tau = \text{Meta}(A)$ requires care: if A is not yet bound, the type of x in Γ' is unresolved. Any lookup of x in the body then yields $\text{Meta}(A)$, which appears in a type relation test. By Lemma 5, a **Meta** node is never the sole cause of a **Lost** judgment — it can always be bound consistently unless the other side is already a conflicting **Exact**. Hence no spurious **Lost** is returned.

Store Exactness

Proposition 1 (Store exactness). *Each store operation (init, resolve, finalize) is a denotationally exact filter: it maps $\mathcal{D}(\mathcal{G})$ to exactly the set of groundings consistent with the new evidence, introducing no ghost models.*

Proof. We verify each operation.

init. The initial obligation graph allocates one evidence node per binding position, with no constraint instances. The denotation is therefore the full set of groundings over the initial node set: no constraint is imposed and no grounding is excluded. The store starts exact.

resolve(Ω, k, ν). Resolving position k with child node ν writes the type evidence E_ν of ν into the obligation ω_b at that position, adding the constraint $E_b = E_\nu$ (or $E_b \subseteq E_\nu$) to the graph. By Lemma 5 this is an exact intersection: $\mathcal{D}(\mathcal{G} \cup \{E_b \sim E_\nu\}) = \{\theta \in \mathcal{D}(\mathcal{G}) \mid \theta \models E_b \sim E_\nu\}$. No grounding satisfying the old store is removed unless it genuinely conflicts with E_ν ; no new grounding is introduced. The store remains exact.

Shared meta-variables. If $\text{Meta}(A)$ appears in two premises φ_1 and φ_2 , obligation locality (Lemma 2) ensures that both premises reference the *same* node in the evidence graph. The constraints from φ_1 and φ_2 are therefore edges on a single shared node, not independent constraints on separate copies. The intersection is computed once over that node, so no ghost model arises from a meta-variable appearing to be satisfiable in each premise independently while being contradictory jointly.

finalize(**Exact**, ...). Exact finalization requires all obligation evidence to be **Exact** and all type constraints to be **Satisfied**. This is a decidable check over closed types (Lemma 5) and is exact: it accepts a grounding if and only if it satisfies every constraint in the current store.

finalize(**Prefix**, ...). Prefix finalization requires the current store to be satisfiable with some closed extension of open evidence. The obligation store at this point is a finite CSP over closed types: each open node ranges over its completion set (finite and non-empty by definition of **Partial**, or the full type domain for **Meta**), and the constraints are the edges of the evidence graph. CSP satisfiability over a finite domain is decidable, so the check is exact. By Lemmas 4–7, **Lost** is returned only when the CSP is genuinely unsatisfiable; no satisfiable grounding is excluded.

Typing Rules

A typing rule $\gamma \in \Theta$ is a tuple $(\text{name}, M, \text{Bind}, \Phi, \hat{\tau}_c)$ where $\text{name} \in N$ is the attached non-terminal, M is a finite set of rule-local meta-variables, Bind is the set of bindings used, $\Phi = \varphi_1, \dots, \varphi_n$ is the ordered premise list, and $\hat{\tau}_c$ is the conclusion type expression. Rules are written in natural-deduction style:

$$\frac{\varphi_1 \quad \cdots \quad \varphi_n}{\hat{\tau}_c} (\text{name}).$$

Premises are evaluated left to right: a meta-variable solved by an earlier premise may appear in later premises and in the conclusion.

Context Transforms

A typing rule may export a *right-bound context transform* $\nabla : \Gamma \mapsto \Gamma[x : \hat{\tau}]$ in its conclusion, written

$$\frac{\varphi_1 \quad \cdots \quad \varphi_n}{\Gamma \rightarrow \Gamma[x : \hat{\tau}] \vdash \hat{\sigma}} \text{ (name)}.$$

Exact nodes may pass ∇ to their right siblings via right-bound context flow (§4.3). Prefix nodes never export ∇ since their right boundary is not yet fixed. Transforms compose left to right: $\nabla_3 = \nabla_2 \circ \nabla_1$.

Realizability and ideal domain match

Theorem 1 (Typing implementation computes the ideal evaluator). *For every reachable state $\sigma(s)$,*

$$\text{eval}_{\text{impl}}(\mathcal{G}(s)) = \text{eval}(\mathcal{G}(s)).$$

where $\text{eval}(\mathcal{G}(s))$ is the ideal evaluator defined in 2.3.

(Definition ??).

Proof. We verify the three cases of Definition 7.

Satisfied. $\text{eval}_{\text{impl}}$ returns **Satisfied** when all obligations are closed and all type constraints hold. This is exactly the condition $\mathcal{G}(s) \in \text{Closed}$, which defines $\text{eval}(\mathcal{G}(s)) = \text{Satisfied}$.

Lost. $\text{eval}_{\text{impl}}$ returns **Lost** when $\mathcal{D}(\mathcal{G}(s)) = \emptyset$, i.e. the obligation CSP has no solution. We claim this holds if and only if no extension r produces $\mathcal{G}(s \cdot r) \in \text{Closed}$.

($\Rightarrow$): By store exactness (Proposition 1), $\mathcal{D}(\mathcal{G}(s)) = \emptyset$ means no consistent grounding of the current constraint graph exists. Since store operations only add nodes and constraints (the graph is monotone), no extension can produce a grounding where none exists now.

($\Leftarrow$): If no extension closes the graph, then no grounding can be completed; by store exactness the current denotation is empty.

Live. The complement of the two cases above, matching $\text{eval}(\mathcal{G}(s)) = \text{Live}$ by Definition 7.

3.1 Simply Typed Lambda Calculus Example

As a reference point, we implement the simply-typed lambda calculus (STLC) using our type system and grammar definition format.

Syntax The STLC syntax can be expressed in BNF-like notation with rule annotations and bindings:

Identifier	$\rightarrow \mathcal{R}([A-Za-z_][A-Za-z0-9]^*)$	
Variable	$\rightarrow \text{Identifier}[x]$	(VAR)
Type	$\rightarrow \text{Identifier} \mid '(\text{Type})' \mid \text{Type} \rightarrow \text{Type}$	
Lambda	$\rightarrow '\lambda' \text{Identifier}[a] ':' \text{Type}[\tau] ':' \text{Expression}[e]$	(LAMBDA)
Application	$\rightarrow \text{Expression}[l] \text{Expression}[r]$	(APP)
Expression	$\rightarrow \text{Variable} \mid \text{Lambda} \mid \text{Application} \mid '(\text{Expression})'$	

Lambda rule.

$$\frac{\Gamma[a:\tau] \vdash e : B}{\tau \rightarrow B} \text{ (LAMBDA)}$$

Here $\mathcal{M} = \{B\}$, $\mathcal{B} = \{a, \tau, e\}$, the single premise extends the context with the annotated binder, and the conclusion is the arrow type.

Application rule.

$$\frac{\Gamma \vdash l : A \rightarrow B \quad \Gamma \vdash r : A}{B} \text{ (APP)}$$

A is unified between the two premises: whatever function type l resolves to determines the required type of r .

Variable rule.

$$\frac{x \in \Gamma}{\Gamma(x)} \text{ (VAR)}$$

Here $\mathcal{M} = \emptyset$, $\mathcal{B} = \{x\}$, the premise is a typing judgment for a variable, and the conclusion is its type.

Partial examples

Valid If we feed the prefix $\lambda x : \text{Int}$, the parser produces a prefix node for the lambda production. The type child is on the right frontier, so the lambda premise is unknown but not contradictory. The oracle accepts a token that extends the type, for example $\rightarrow \text{Bool}$, and it also accepts a $.$ token that closes the annotation and starts the body.

Invalid If we feed the prefix $\lambda f : \text{Int} \rightarrow \text{Bool}. f (\lambda y : \text{Int}. y ($, the final parenthesis leaves the term syntactically open, but the semantic contradiction is already stable. The variable f has type $\text{Int} \rightarrow \text{Bool}$, so its argument must have type Int . The argument has already committed to the lambda alternative, so any successful completion of that subtree would have arrow shape $\text{Int} \rightarrow B$ for some body type B . No future token can turn a lambda into an integer argument, so the oracle rejects the prefix without waiting for the closing parenthesis.

4 Prefix Parsing Algorithm

The core algorithm is a language-parametric prefix parser. While we developed it for constrained generation, it can also serve as a static analysis tool, a completion engine, or a full type checker.

4.1 Language-Parametric Interface

The framework is parametric over SPGs, materialized in `.auf` specifications: a file defines productions, binding annotations, and semantic rules. The engine checks well-formedness, builds binding maps as in Section 2.2, and then exposes the same prefix oracle for every language instance.

4.2 Parsing

A *prefix parser* for an SPG G is a function

$$\Psi_G : \Sigma^* \rightarrow \mathcal{F}_\partial \cup \{\text{reject}\},$$

It is defined by a runtime state γ that includes :

- The input segmentation $x = \text{seg}_\mathcal{L}(s) = (x_0, \dots, x_{m-1})$
- The agenda $\mathcal{A}$ of items to process
- The chart C mapping recognized nonterminal spans to node identifiers
- The waiter relation W mapping nonterminal spans to paused parents
- The set R of known end positions for each nonterminal and segment index
- The end-of-input frontier F of items awaiting EOF closure
- The packed parse arena H

A live item is

$$\eta = \langle n, p, k, [i, j], \Gamma, \Omega, \chi \rangle$$

where n is the nonterminal being recognized, p is one of its productions, k is the dot position, $[i, j]$ is the consumed segment span, Γ is the semantic context at the dot, Ω is the current obligation store, and χ is the ordered list of accepted child references and terminal evidence.

$$p = \underbrace{\alpha_1[b_1] \cdots \alpha_k[b_k]}_{\text{recognized}} \bullet \underbrace{\alpha_{k+1}[b_{k+1}] \cdots \alpha_n[b_n]}_{\text{remaining}}$$

Each arena node summarizes a family of parses by

$$\nu = \langle n, [i, j], \rho, \tau, \nabla, \beta \rangle$$

where n is a nonterminal, $\rho \in \{\text{Exact}, \text{Prefix}\}$ is the node status, τ is semantic type information, ∇ is an optional exported context transform, and β is parent-visible binding evidence. Exact nodes are closed on the current input and may export ∇ while prefix nodes are open at the right frontier and never export context transforms.

Semantic Interface The parser uses the following semantic operations. They are partial: failure means that the branch is discarded.

- $\text{init}(\mathcal{T}(n), \Gamma_0)$: initial obligations for n ,
- $\text{descend}(n, p, k, \Gamma, \Omega, \chi)$: context used to enter the child after the dot,
- $\text{step}(n, p, k, \Omega)$: obligations projected to that child,
- $\text{resolve}(\Omega, k, \nu)$: obligations after accepting child node ν ,
- $\text{finalize}(\rho, n, p, \Gamma, \Omega, \chi)$: semantic summary for an exact or prefix node.

Exact finalization requires **Now** and may export ∇ . Prefix finalization requires **Ext** and exports $\perp$. We assume all semantic operations are denotational filters: defined outputs denote exactly the satisfying refinements, and semantic rejection happens only on empty denotation.

Parse Inference The following rules summarize the semantic fields added to the standard prefix-Earley transition system. They omit agenda, chart, arena interning, and standard Earley side conditions such as completed-dot checks: deriving an item or node means inserting it into the corresponding runtime structure if it has not already been seen.

$$\begin{array}{c}
\frac{p \in A(S) \quad \Omega_0 = \text{init}(\mathcal{T}(S), \Gamma_0)}{\langle S, p, 0, [0, 0], \Gamma_0, \Omega_0, \epsilon \rangle} \text{SEED} \\
\\
\frac{\eta = \langle n, p, k, [i, j], \Gamma, \Omega, \chi \rangle \quad p[k] = n' \quad n' \in N \quad q \in A(n')}{W(n', j) \ni \langle n', q, 0, [j, j], \text{descend}(n, p, k, \Gamma, \Omega, \chi), \text{step}(n, p, k, \Omega), \epsilon \rangle} \text{PREDICT} \\
\\
\frac{\eta = \langle n, p, k, [i, j], \Gamma, \Omega, \chi \rangle \quad p[k] = t \quad t \in T \quad \text{match}(t, x_j) = \xi \quad \text{closed}(\xi)}{\langle n, p, k+1, [i, j+1], \Gamma, \Omega, \chi \cdot \text{Term}(x_j, \xi) \rangle} \text{SCANEXACT} \\
\\
\frac{\eta = \langle n, p, k, [i, j], \Gamma, \Omega, \chi \rangle \quad p[k] = t \quad t \in T \quad j = m-1 \quad \text{match}(t, x_j) = \xi \quad \text{open}(\xi)}{F \ni \langle n, p, k+1, [i, m], \Gamma, \Omega, \chi \cdot \text{Term}(x_j, \xi) \rangle} \text{SCANOPEN} \\
\\
\frac{\eta \quad \text{finalize}(\text{Exact}, n, p, \Gamma, \Omega, \chi) = (\tau, \nabla, \beta)}{\nu = \langle n, [i, j], \text{Exact}, \tau, \nabla, \beta \rangle} \text{COMPLETEEXACT} \quad \frac{\eta \in F \quad \text{finalize}(\text{Prefix}, n, p, \Gamma, \Omega, \chi) = (\tau, \perp, \beta)}{\nu = \langle n, [i, m], \text{Prefix}, \tau, \perp, \beta \rangle} \text{COMPLETEPR} \\
\\
\frac{\langle n, p, k, [i, j], \Gamma, \Omega, \chi \rangle \in W(n', j) \quad \nu = \langle n', [j, \ell], \text{Exact}, \tau, \nabla, \beta \rangle \quad \Omega' = \text{resolve}(\Omega, k, \nu)}{\langle n, p, k+1, [i, \ell], \nabla(\Gamma), \Omega', \chi \cdot \nu \rangle} \text{RESUMEEXACT} \\
\\
\frac{\langle n, p, k, [i, j], \Gamma, \Omega, \chi \rangle \in W(n', j) \quad \nu = \langle n', [j, m], \text{Prefix}, \tau, \perp, \beta \rangle \quad \Omega' = \text{resolve}(\Omega, k, \nu)}{F \ni \langle n, p, k+1, [i, m], \Gamma, \Omega', \chi \cdot \nu \rangle} \text{RESUMEPREFIX}
\end{array}$$

The parser succeeds when saturation and EOF closure produce a root node

$$\nu = \langle S, [0, m], \rho, \tau, \nabla, \beta \rangle \quad \rho \in \{\text{Exact}, \text{Prefix}\}.$$

If no such root node is produced, the parser returns **reject**.

4.3 Context flow

The context stored in an item is the context at the dot. It is obtained by combining two forms of context flow: top-down context passed into a child, and right-bound context exported by exact children to their right siblings.

Downwards context When the parser predicts a child at position k , it computes the child context by

$$\Gamma_{\text{child}} = \text{descend}(n, p, k, \Gamma, \Omega, \chi).$$

This is a downward flow: the child is parsed under the context selected by the parent rule. For example, the body of a lambda is predicted under $\Gamma[x : \tau]$, but this extension is local to that child. It is a property inherited from recursive descent, where its triviality is more apparent.

Right-bound flow Right-bound flow happens after a child is completed. If exact children to the left of the dot export transforms

$$\nabla_1, \dots, \nabla_q,$$

then the context at the dot is

$$\Gamma_q = (\nabla_q \circ \dots \circ \nabla_1)(\Gamma_0).$$

Accepting the next exact child with export ∇_{q+1} updates the parent context to

$$\Gamma_{q+1} = \nabla_{q+1}(\Gamma_q) = (\nabla_{q+1} \circ \nabla_q \circ \dots \circ \nabla_1)(\Gamma_0).$$

Prefix nodes do not participate in this composition: their export is $\perp$, so resuming a prefix child leaves the parent context unchanged. Thus top-down flow determines the context used inside a child, while right-bound flow determines the context seen by later siblings.

Definition 8 (Witnessed liveness). *Let q be an item or node retained after parsing a prefix s . A witness for q is a pair (r, θ) , where $r \in \Sigma^*$ and θ is a closed semantic valuation, such that:*

1. *the syntactic projection of q completes on the input sr ;*
2. *the completion of q along that branch realizes the valuation θ ;*
3. *θ satisfies the semantic proof graph carried by q .*

We say that q is witness-live when it has such a witness.

4.4 STLC Engine Trace

We illustrate the parser on the complete STLC term

$$\lambda x : \text{Int}. x$$

under the empty initial context Γ_0 . We focus on the lambda alternative p_λ and use the typing rules from Section 2.

Step	Parser event	Semantic effect
1	Seed lambda alternative	create obligations for a, τ, e and output $\tau \rightarrow B$
2	Scan $\lambda x : \text{Int}$.	resolve $a = x, \tau = \text{Int}$
3	Predict body e	enter body under $\Gamma_1 = \Gamma_0[x : \text{Int}]$
4	Complete body variable x	infer body type $\Gamma_1(x) = \text{Int}$
5	Resume lambda	unify $B = \text{Int}$, output $\text{Int} \rightarrow \text{Int}$

Table 1. Compact semantic trace for parsing $\lambda x : \text{Int}.x$.

Semantic details The key step is Step 3. After the binder and annotation have been scanned, the lambda rule turns the body premise

$$\Gamma[a : \tau] \vdash e : B$$

into

$$\Gamma_1 \vdash e : B, \quad \Gamma_1 = \Gamma_0[x : \text{Int}].$$

Thus the body is not parsed under the outer context Γ_0 , but under the child-bound context introduced by the lambda binder. When the body variable x completes, the variable rule gives

$$\Gamma_1(x) = \text{Int}.$$

Resuming the lambda item discharges the body obligation by unifying $B = \text{Int}$ so the completed lambda has type $\text{Int} \rightarrow \text{Int}$.

4.5 Soundness

The parser is used as a prefix oracle. The property we need is that every accepted prefix can still be completed into a closed semantic parse. Here Ψ_G denotes the saturated inference system defined above: every item or node whose syntactic side conditions hold and whose semantic operation is defined is eventually inserted.

We use two invariants of the rules above. First, after erasing semantic fields, the rules form the standard Earley prefix parser for the syntactic projection of G . Second, semantic operations are exact local filters: a semantic operation is defined exactly when the evidence graph carried by the current item or node is not lost. In particular, exact finalization requires a closed satisfying graph, while prefix finalization requires a live graph in the sense of Definition 7.

Theorem 2 (Completeness soundness). *Let $G \in \text{SPG}(D)$ be productive by 1. If*

$$\Psi_G(s) \neq \text{reject},$$

then there exists $s' \in \Sigma^$ such that $\Psi_G(ss')$ produces an Exact root node.*

Proof. Since $\Psi_G(s) \neq \text{reject}$, saturation on s produced a root node

$$\nu = \langle S, [0, m), \rho, \tau, \nabla, \beta \rangle$$

with $\rho \in \{\text{Exact}, \text{Prefix}\}$.

If $\rho = \text{Exact}$, the root is already closed on the current input, so taking $s' = \varepsilon$ proves the result.

Suppose instead that $\rho = \text{Prefix}$. This root was produced by prefix finalization. By the semantic side condition of that rule, the evidence graph carried by this root is live. By Definition 7, there exists a suffix $s' \in \Sigma^*$ that closes this same graph into a satisfying one.

It remains only to justify that the parser will actually produce the closed root on ss' . The suffix s' closes the branch summarized by ν . Syntactically, the erased inference system is the Earley prefix parser, so the corresponding syntactic continuation is admitted. Semantically, each operation on that branch is an exact filter, so following a satisfying continuation cannot make a required semantic operation fail. Since the system is saturated, all items and nodes on this closed branch are eventually inserted. In particular, $\Psi_G(ss')$ produces an Exact root node.

Lemma 8 (Prefix completeness of saturation). *Let $G \in \text{SPG}(D)$. If $\Psi_G(sr)$ produces an Exact root node, then*

$$\Psi_G(s) \neq \text{reject}.$$

Proof. Take a saturated derivation of the exact root over sr , and cut it at the input frontier after s . By prefix-awareness of the segmentation policy, closed interior segments are unchanged by this cut; only the final segment may become open.

The syntactic projection of the cut derivation is therefore a valid prefix Earley derivation for s . Semantic evidence already to the left of the cut is unchanged. Evidence crossing the cut becomes prefix evidence, and it is retained because the complete derivation over sr gives an actual suffix closing it.

Moreover, semantic graphs grow monotonically with input:

$$\mathcal{E}(s) \subseteq \mathcal{E}(sr) \quad \text{and} \quad \mathcal{C}(s) \subseteq \mathcal{C}(sr).$$

The exact root over sr has a closed satisfying graph. Restricting that closed graph to the evidence and constraints already present at s gives a satisfying grounding for the prefix graph. Hence none of the semantic operations in the cut derivation is lost.

Since the inference system is saturated, the corresponding exact or prefix root node is inserted when parsing s . Therefore

$$\Psi_G(s) \neq \text{reject}.$$

Theorem 3 (Prefix monotonicity). *Let $G \in \text{SPG}(D)$ be productive. If*

$$\Psi_G(s) \neq \text{reject},$$

then for every prefix s'' of s ,

$$\Psi_G(s'') \neq \text{reject}.$$

Proof. Let $s = s''r$. By Theorem 2, there exists $t \in \Sigma^*$ such that

$$\Psi_G(st) = \Psi_G(s''rt)$$

produces an Exact root node. Applying Lemma 8 to the complete input $s''rt$ gives

$$\Psi_G(s'') \neq \text{reject}.$$

5 P7 Constrained Generation

P7 is the model-facing layer that uses the AUFBAU prefix oracle during decoding. The shared core does not depend on any particular language model: the LLM proposes continuations, and the semantic prefix oracle removes continuations that would leave the live syntactic and semantic prefix space.

5.1 System Architecture

Figure 2 shows the overall system architecture and generation loop. Unlike logit-filtering approaches, P7 does not ask AUFBAU for a full admissible vocabulary before sampling. Instead, sampling happens on the GPU. The sampled token is then checked by AUFBAU using `try_feed`. If the token is rejected, P7 masks that token for the current step and resamples. This approximates sampling from a distribution containing allowed elements only but has better average complexity, since in most cases, the good token will be among the highest if the model has some language data.

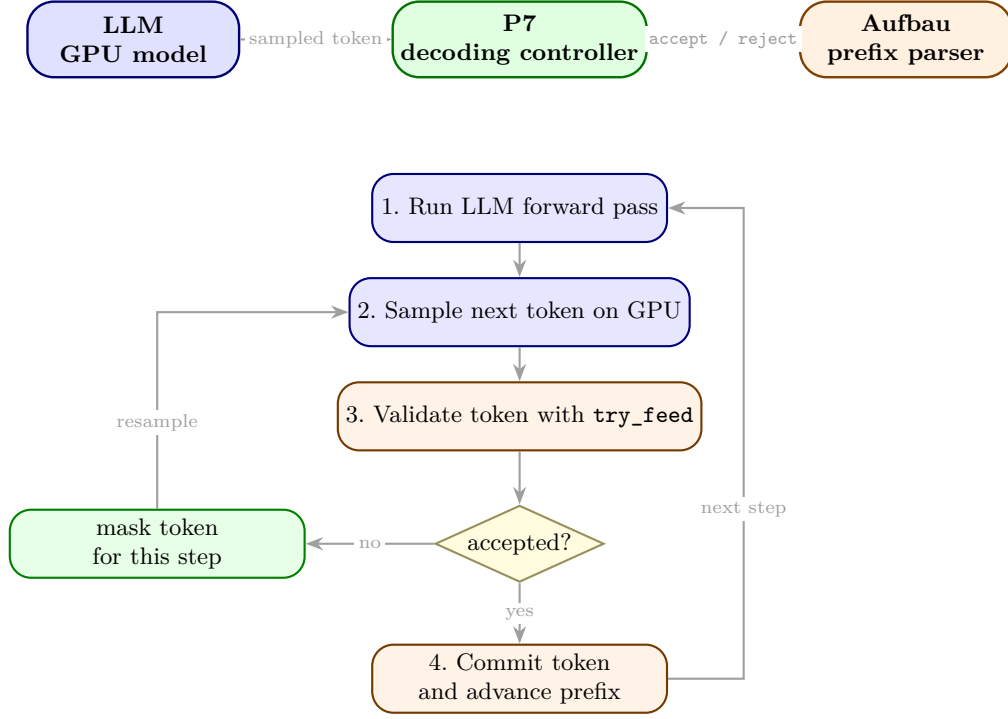


Fig. 2. P7 system structure and constrained decoding loop. The language model proposes tokens, P7 validates them with Aufbau through `try_feed`, rejected tokens are masked for the current step, and accepted tokens are committed to the prefix.

5.2 Constrained Decoding

The constrained decoding system, implemented in P7, integrates AUFBAU with Hugging Face Transformers [19]. The generation loop is illustrated in Figure 2.

At each decoding step:

1. The LLM computes logits for the next token.
2. A token is sampled on the GPU from the current distribution.
3. P7 passes the sampled token to AUFBAU using `try_feed`.
4. If AUFBAU accepts the token, P7 commits it and advances the prefix.
5. If AUFBAU rejects the token, P7 masks that token for the current step and resamples.

Thus P7 uses AUFBAU as an online validation oracle rather than as a precomputed vocabulary filter. This avoids constructing the full admissible token set at every step. The decoder only queries the semantic parser for tokens that the model actually samples. Deterministic decoding can be implemented by sampling from a degenerate distribution or by repeatedly selecting the highest unmasked token and checking it with `try_feed`.¹

¹ While not the most efficient, and maybe too probabilistic for some applications, this approach avoids copying back and forth between CPU and GPU for the full vocabulary.

Token-level interface and termination The formal parser is defined over character strings, while a model emits tokens from a finite tokenizer vocabulary V . At a decoding state with committed text s , a token $a \in V$ is interpreted by its decoded text fragment $\text{dec}(a)$. The call `try_feed(a)` accepts exactly when

$$\Psi_G(s \text{dec}(a)) \neq \text{reject}.$$

Thus token validation is reduced to the same character-level prefix oracle used in the formal semantics. The tokenizer layer does not need to precompute the full admissible vocabulary; it only asks whether the sampled token’s decoded string keeps the current text in the semantic prefix language. The regular-terminal completeness machinery used for this bridge follows the prefix-completion construction described in [6].

The resampling loop terminates because V is finite. During one decoding step, P7 maintains a masked set $M \subseteq V$. A rejected token is inserted into M and cannot be sampled again at that step. The loop stops either when some token is accepted and committed, or when $M = V$. In the latter case all tokens, including any end-of-sequence token present in V , have been rejected for the current prefix, so generation reports that the state is blocked rather than continuing to resample.

5.3 Proof-Search Graph Perspective

The decoding process can be read as graph search over semantic parser states. Nodes are parser configurations together with outstanding semantic obligations; edges are token or tree-expansion choices; accepting an edge means the target state still has a semantic prefix parse. The language model supplies a probability distribution over outgoing edges, while AUFBAU removes edges whose targets contain stable syntactic or semantic contradiction. This does not prove that the model selects the best proof or program, but it ensures that the generated trace remains inside the semantically live portion of the search graph.

6 Evaluation

The evaluation separates oracle validation from model-facing generation. The first part checks that the implementation follows the semantic prefix model; the second part measures what the oracle changes when placed in the P7 decoding loop.

6.1 Oracle Validation

In order to evaluate the correctness of the AUFBAU engine during development, we built a validation suite that tests the completeness property from Theorem 2. We take a string $s = c_0 \cdots c_n$, $c \in \Sigma$, and we ensure that every prefix $c_0 \cdots c_k$, $k \leq n$ is accepted by the parser. We also *completion monotonicity* by running an AUFBAU backed program synthesis job on each prefix, and ensuring the number of found complete completions decreases as k increases.

6.2 Constrained vs. Unconstrained Generation

To evaluate the real-world performance benefits of constrained generation in specific program synthesis task, we compare across open models up to 31 billion parameters, the success rate on a custom benchmark mixing simple instruction-following, coding tasks and logic problems. Our benchmark

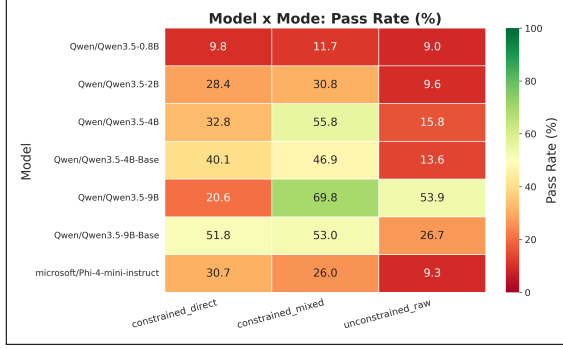


Fig. 3. Matrix heatmap showing constraint satisfaction across different model sizes and grammar types.

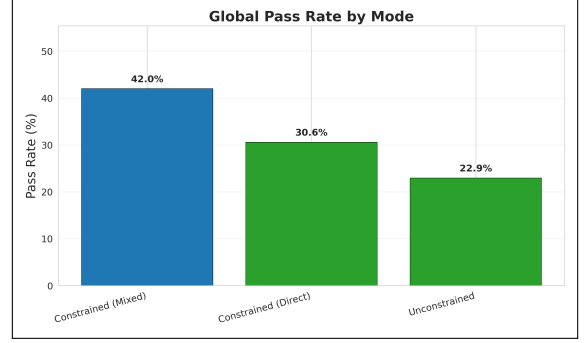


Fig. 4. Pass rate comparison across constrained and unconstrained generation

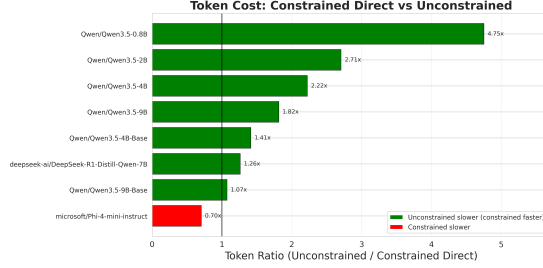


Fig. 5. Token efficiency by mode

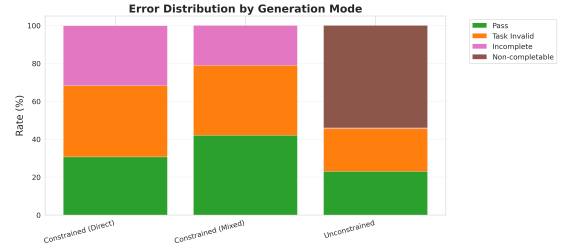


Fig. 6. Error repartition by mode

tasks use the **STLC**, **IMP** and **FUN** languages. In order to validate, we first ensure the program is complete and well-typed, then, using a small interpreter we implemented in python, and three task resolution models we validated for task-validity:

- We check for program equivalence between the LLM produced result and an expected code string, normalizing with β -reduction and α -reduction in the **FUN** and **STLC** cases. We check for type and structural correspondence.
- We match the tput against a series of task defined tests that it must pass to be considered correct. In the case of factorial, the resolution was computing factorials up to n . With some komolgorov complexity approximation we know that in the given token restrcition the LLM could not have possibly cheated.
- We set an expected context state, and match it up against the produced context after executing the LLM-produced code.

In the suite, we have 3 decoding modes:

1. **Unconstrained (raw)** meaning the model must generate directly the expected output in a N token limit
2. **Constrained (direct)** where the models must generate directly the expected output, but under P7 constraints, in the same N token limite

3. **Constrained (mixed)** where we mix a unconstrained generation streams for M token, and a formally constrained output block for N tokens.

Alls the models have the exact same prompt, containing grammar details, examples, and clear instructions about the tasks. As a reference point big models like GPT-5.3 managed to pass 100% of the tests.

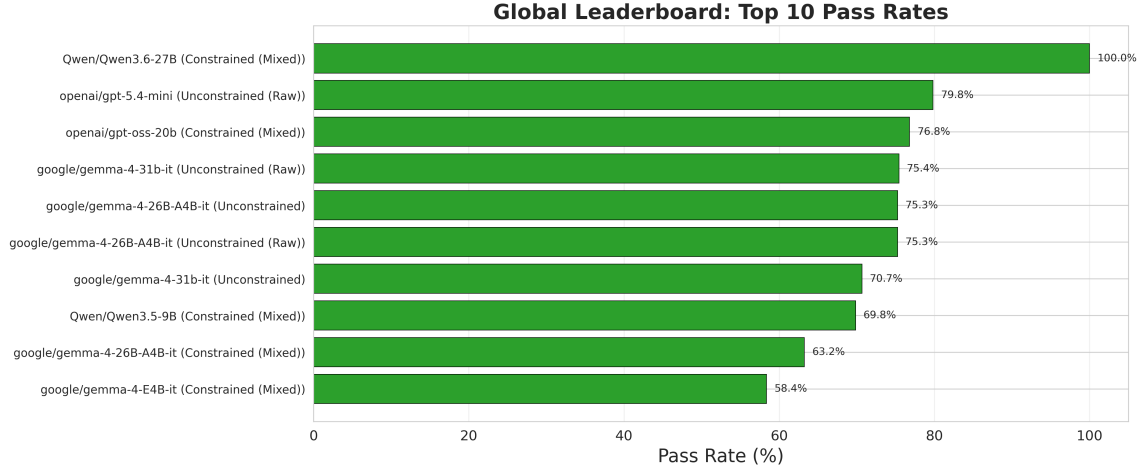


Fig. 7. Top 10 scores on benchmarks with both constrained and unconstrained models

Leaderboard To see how constrained generation help smaller models fare against bigger ones, we tried the unconstrained workflow on closed source and bigger models, using the OpenRouter API (GPT-5.4-mini, gemma-4-31b-it). As a result, we see an impressive performance of smaller open-models under P7, with Qwen-3.6-27B surpassing GPT-5.4-mini.

ID	Prompt	Initial Prefix
stlc_apply_twice_int	Apply function twice to Int	<code>λ f:(Int->Int).</code>
imp_while_factorial_four	Compute factorial using while loop	<code>{ let n: Int =</code>
fun_compose	Compose two functions	<code>let compose:</code>

Table 2. Representative examples of benchmarks tasks

Limitations and real-world targets While the results are impressive, we can't assume they fully port to general programming tasks, as we use specific languages such as FUN, IMP, or even STLC which is probably underrepresented in coder models.

However, it is much closer to tool-use in agentic workflows, or API interaction. Indeed, in those settings, the model has to interact with conventions and languages that are possibly out of the training data, and might include context dependent constraints.

This is why such environment are a better target for our system. Contrary to Mundler et al. [CITE](#) or [CITE other programming direct constrained gen](#), this system isn't made for long program generation task, but rather for varied interactions with specific formal systems.

7 Related Work

Formal language foundations. Our syntactic core is classical: Earley parsing gives a sound and complete dynamic-programming recognizer for CFGs [3], and packed forests/GLR-style sharing are standard tools for ambiguous grammars [16]. The semantic layer is grounded in attribute grammars and syntax-directed definitions [5]: productions expose child evidence, and rules compute semantic facts from that evidence. Constraint and logic grammar formalisms, including definite-clause grammars [11] and unification-based grammars [14], similarly accept derivations by solving attached constraints. Our class should be read as a decidable, prefix-conservative fragment of this lineage. The difference is the online setting: attribute and constraint evaluation is usually defined over complete trees; here the evaluator must be conservative over partial forests and right-frontier evidence.

Grammar-constrained decoding. PICARD [13] popularized incremental parsing as a token rejection mechanism for text-to-SQL. Outlines [18] and later grammar-constrained systems enforce regular or context-free structure during generation. DOMINO [1] and Park et al. [10] sharpen the tokenizer-alignment problem and show how much efficiency depends on aligning subword vocabularies with grammar terminals. Formatron [15] is the closest systems baseline for our parser architecture: it uses Earley-driven dynamic pruning, caching, and related engineering optimizations for efficient structured decoding. The distinction is semantic: Formatron targets syntactic structure such as regexes, CFGs, and JSON schemas, whereas our oracle carries prefix-conservative semantic obligations, binding evidence, and context transforms. These systems provide the syntactic baseline that our semantic prefix oracle generalizes.

Semantic and type constraints. Synchromesh [12] constrains code generation with syntax, scope, typing, and contextual logic in concrete domains. Mündler et al. [7] give a type-constrained decoder based on prefix automata and inhabitable type search, closest to our type-directed instance. ChopChop [8] is closest in spirit at the semantic level, but the main differences are our constraint targets: while ChopChop implements generic constraints over an output space, we offer a drop-in framework to define *languages*.

Structured output systems. JSON-schema and structured-output benchmarks [4] show that constrained decoding is becoming infrastructure, not a niche technique. Systems such as SGLang [20] expose structured generation as a programming interface. Our work targets the next layer: outputs whose validity depends on semantic state accumulated during parsing, not only on schema shape.

8 Limitations

- **Language coverage:** the current semantic algebra supports finite typing contexts, local unification, binding lookup, and context transforms. Richer TypeScript or Python fragments require additional premise forms and context domains.
- **Performance:** while overhead is manageable for typical grammars, large grammars with many typing rules and complex nesting can slow decoding.

9 Conclusion

We presented semantic prefix grammars, a language-parametric framework for constrained generation with context-dependent semantic checks over prefix parse forests. The core idea is to keep syntax ordinary and make semantics explicit: bound productions expose evidence, semantic premises evaluate conservatively over partial input, and exact nodes alone may export context effects. This gives a principled path from CFG-only constrained decoding toward realistic programming-language fragments, even beyond the current implemented premise core. Key directions for future work are richer semantic algebras for TypeScript and Python, mechanized proofs of the semantic invariants, and tighter integration with tokenizer-level masking and model training.

Acknowledgments

We thank Niels Mündler at ETH Zürich’s SRI Lab for collaboration and feedback on the theoretical framework and verification pipeline. This work was supported by Unsuspicious Industries, an independent nonprofit research collective.

References

1. Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Guiding LLMs the right way: Fast, non-invasive constrained generation. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pages 3658–3673, 2024.
2. Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
3. Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
4. Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. Generating structured outputs from language models: Benchmark and studies. *arXiv preprint arXiv:2501.10868*, 2025.
5. Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
6. Paul Kronlund-Drouault. Completeability and regular expressions. <https://unsuspicious.org/blog/completing-regex/>, 2025. Unsuspicious Industries, Oct. 12, 2025.
7. Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. Type-constrained code generation with language models. *Proceedings of the ACM on Programming Languages*, 9(PLDI), 2025.
8. Shaan Nagy, Timothy Zhou, Nadia Polikarpova, and Loris D’Antoni. Chopchop: A programmable framework for semantically constraining the output of language models. In *Proceedings of the 53rd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2026.

9. Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190, 2009.
10. Kanghee Park, Timothy Zhou, and Loris D’Antoni. Flexible and efficient grammar-constrained decoding. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, pages 48262–48275, 2025.
11. Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, 1980.
12. Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.
13. Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9895–9901. Association for Computational Linguistics, 2021.
14. Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. Number 4 in CSLI Lecture Notes. CSLI, Stanford, CA, 1986.
15. Xintong Sun, Chi Wei, Minghao Tian, and Shiwen Ni. Earley-driven dynamic pruning for efficient structured decoding. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025. arXiv:2506.01151.
16. Masaru Tomita. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 455–462, 1985.
17. Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.
18. Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023.
19. Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumont, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020.
20. Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024.

A Mixed generation results

While mixed generation has shown the best results overall, we still observe weird patterns in the benchmarks results. Most models benefited from the CoT space, but other have been disadvantaged. This might be linked to the sampling methods, or to a CoT cutoff due to a limited think token budget.

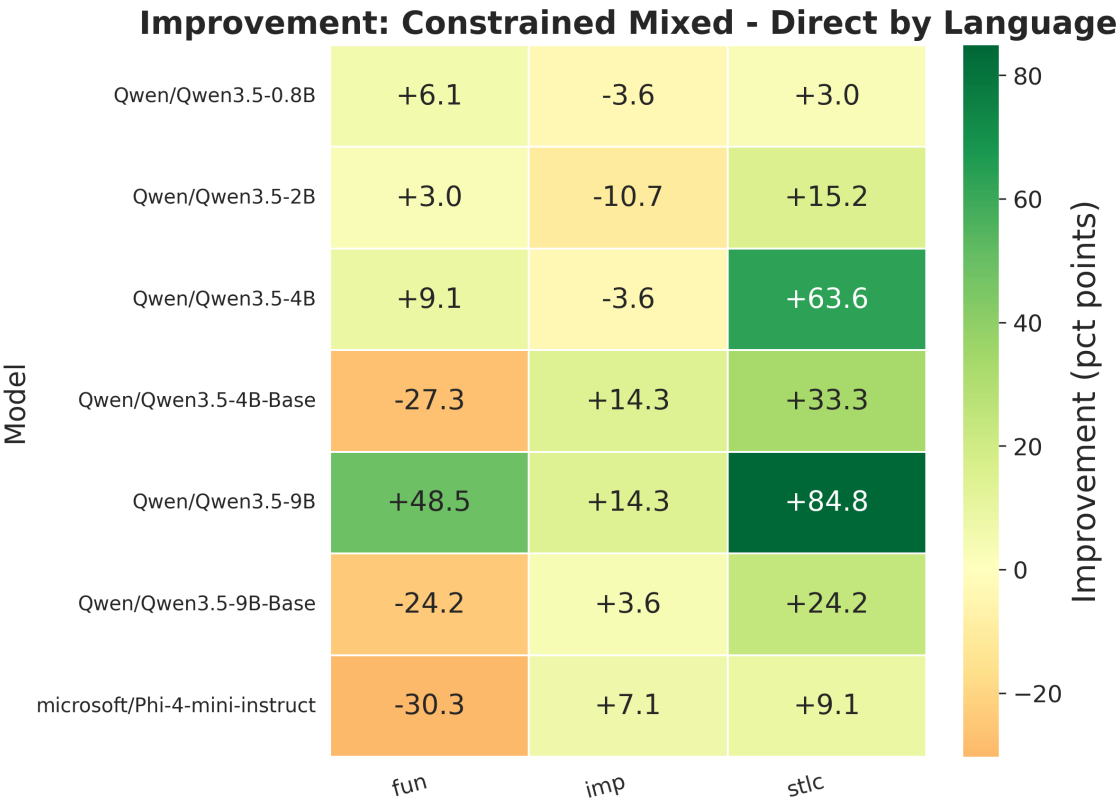


Fig. 8. Percentage point improvement between direct constrained generation and mixed mode