Generalized LL Decoding with Context-Sensitive Constraints

Anonymous ACL submission

Abstract

We propose *generalized LL decoding (GLLD)*, a constrained decoding algorithm that enables an autoregressive model to sequentially generate tokens as terminals whose concatenation satisfies a formal grammar with context-sensitive constraints. To address context-sensitive constraints, we augment a CFG with predicates, mutators and selectors. The augmented grammar is supported by our generalized LL (GLL) parser, whose data-structures are modified with linked-list nodes, lookaheads and environments for constrained decoding. For efficiency, we apply right-recursion optimization to our GLL parser while preserving context-sensitive constraints. Our parser is combined with beam search to generate high-likelihood terminal sequences that are valid with respect to our grammar in polynomial time. We apply GLLD to semantic parsing on the NLVR benchmark, where decoding with context-sensitive constraints improves accuracy and consistency, while maintaining fast decoding speed.

1 Introduction

Constrained decoding is a method that guides an autoregressive model to generate a token sequence that satisfies given constraints. The constraints can be defined from context-free grammars (CFGs) (Yin and Neubig, 2017; Krishnamurthy et al., 2017; Wu et al., 2021; Shin et al., 2021), knowledge bases (Chen et al., 2021; Cao et al., 2021; Lee and Shin, 2025; Nam and Lee, 2026), or variable bindings (Krishnamurthy et al., 2017; Poesia et al., 2022). Constrained decoding methods are especially crucial for tasks that need strict output syntax, such as semantic parsing or general-purpose code generation.

Various parsing algorithms have been used for constrained decoding to ensure context-free constraints. Tuccio et al. (2025) use $\mathrm{LL}(1)$ parsers (Knuth, 1965), Wu et al. (2021), Baranowski and Hochgeschwender (2021), and Ugare et al. (2025) use $\mathrm{LR}(1)$ parsers (Knuth, 1965), Shin et al. (2021), Moskal (2024), and Li et al. (2026) use Earley parsers (Earley, 1970), and Koo et al. (2024) and Dong et al. (2025) use a parser based on pushdown automata (Oettinger, 1961). The constrained decoding methods use the parsers to check whether a prefix token sequence is valid with respect to a CFG, or to get all valid tokens from the current parsing state.

In particular, the Earley parsing algorithm is popular for constrained decoding, as the algorithm can perform parsing for a general CFG in polynomial time, but the algorithm cannot efficiently process a formal grammar with context-sensitive constraints. A data-dependent grammar (Jim et al., 2010) enables an Earley parser to process context-sensitive constraints, but failure of a predicate in the right-hand side of a production rule makes the parser back-track, then an autoregressive model is forced to re-predict tokens from a previous time step, resulting in a critical reduction in efficiency. Alternatively, the framework of predicates and mutators in $\mathrm{LL}(*)$ (Parr and Fisher, 2011) can be applied to an Earley parser, as the Earley parser operates in a top-down manner, but the Earley parser should sacrifice optimization methods for ε-production rules (Aycock and Horspool, 2002) and right-recursive production rules (Leo, 1991).

In this paper, we propose the generalized LL decoding (GLLD) algorithm that sequentially generates tokens as terminals whose concatenation follows a formal grammar with context-sensitive constraints. To address context-sensitive constraints, we extend a CFG with predicates and mutators (Parr and Fisher, 2011) and selectors. For parsing with the grammar, we modified a generalized LL (GLL) parser (Scott and Johnstone, 2009, 2013; Afroozeh and Izmaylova, 2015a) for efficient constrained decoding.

Our contributions are listed as follows:

1. We extend a CFG with predicates and mutators (Parr and Fisher, 2011) and selectors to address context-sensitive constraints (Section 3.1).
2. We modify the data structures for a GLL parser variant (Afroozeh and Izmaylova, 2015a) with linked list nodes to allow multiple hypotheses, with lookaheads to keep predicted terminals, and with environments to store context-sensitive constraints (Section 3.2).
3. We optimize right recursion (Leo, 1991) in our GLL parser by prioritizing descriptors (Section 3.2), while strictly preserving the context-sensitive constraints (Section 3.3).
4. We devise the GLLD algorithm (Section 3.5) that combines the beam search algorithm with our GLL parser (Section 3.4) to generate valid terminal sequences with respect to our grammar, which treats tokens as terminals, in polynomial time (Section 3.6).
5. We applied the GLLD algorithm to the semantic parsing task (Section 3.7).

2 Background

GLL parsing (Scott and Johnstone, 2009, 2013; Afroozeh and Izmaylova, 2015a) is a top-down parsing algorithm that generalizes the $\mathrm{LL}(k)$ parsing algorithm to handle general CFGs. A GLL parser replaces the stack of an $\mathrm{LL}(k)$ parser with a graph-structured stack (GSS), and stores partial parse trees in a shared packed parse forest (SPPF), then the data structures enable polynomial time parsing. The worst-case time complexity of GLL parsing with a GSS and a SPPF is $O(|\zeta(P)|\, n^3)$ for a general CFG, where $|\zeta(P)|$ is the sum of the lengths of all production rules' right-hand sides. However, in constrained decoding, lookaheads are not available until they are predicted, so filtering production rules with FOLLOW for efficiency is impossible. To improve reduction efficiency, we adapt the right recursion optimization method (Leo, 1991) into our GLL parser (Section 3.3).

Earley parsing (Earley, 1970) is a parsing algorithm that handles general CFGs, and optimization methods for the algorithm has been developed. Leo (1991) proposed the right recursion optimization that prevents an Earley parser from repeatedly taking the completer operation for a right-recursive rule if a transitive item exists. Aycock and Horspool (2002) modified the predictor to pass nullable non-terminals to prevent the problem of ε-production rules. However, with context-sensitive constraints, a non-terminal can be conditionally nullable, and this case makes the optimization methods are hard to apply. To apply the right recursion optimization method (Section 3.3), we prioritize descriptors (Section 3.2).

Context-sensitive constraints have been incorporated into parsing algorithms. Data-dependent grammars have been applied to Earley parsing (Jim et al., 2010) and GLL parsing (Afroozeh and Izmaylova, 2015b). In particular, Afroozeh and Izmaylova (2015b) inspired us to augment the data-structures of a GLL parser with linked-list nodes, lookaheads and environments for constrained decoding. However, predicates in the right-hand side of a production rule can cause a parser to back-track and an autoregressive model to re-predict tokens from a previous time step. Therefore, we adapt the framework of predicates and mutators in $\mathrm{LL}(*)$ parsing (Parr and Fisher, 2011), in which predicate is used to determine whether a production rule can be applied, and in which a mutator updates an environment for a specific position in the right-hand of a production rule.

3 Method

We describe the GLLD algorithm with respect to grammars (Section 3.1), data structures (Section 3.2), right recursion optimization (Section 3.3), GLL parsing (Section 3.4), constrained decoding (Section 3.5), complexities (Section 3.6), and semantic parsing (Section 3.7). The notation and pseudo-code that is omitted from the main text are listed in Appendices A and B.

3.1 Grammars

We define our grammar G as a 7-tuple $(N, T, P, S, \Pi, M, \Sigma)$, in which the first four components N, T, P and S are inherited from a CFG. N is a set of non-terminals, T is a set of terminals, $P \subset N \times (N \cup T)^*$ is a set of production rules, S is the start non-terminal. In addition, we define a pre-terminal as a non-terminal that is used as the left-hand side of a production rule whose right-hand side should be a single terminal. For example, a pre-terminal A is used as the left-hand side of the rule $A ::= a$ in which a is a terminal. In other words, when a production rule is applied to a pre-terminal, the pre-terminal is replaced as a single terminal. Therefore, N is split into two subsets: N^{G} which is a set of non-terminals that

are not pre-terminals and N^{P} which is a set of pre-terminals.

The remaining three components Π, M and Σ of the grammar G are introduced for context-sensitive constraints. Π is a mapping that assigns a production rule $r \in P$ to a predicate $\pi : L \times E \rightarrow \{\mathrm{true}, \mathrm{false}\}$, M is a mapping that assigns a slot $s \in \zeta(P)$ to a mutator $\pi : L \times E \rightarrow E$, and Σ is a mapping that assigns a pre-terminal $A \in N^{\mathrm{P}}$ to a selector $\sigma : L \times E \rightarrow 2^T$, where L is the set of linked list nodes that store sequences of terminals, E is the set of environments, and $\zeta(P) = \{(r, i) \mid r \in P, i \in \{0, \ldots, |\mathsf{rhs}(r)|\}\}$ is the set of slots, in which a slot $s = (r, i)$ is represented as a production rule $r = A ::= \alpha\ \beta$ with a dot in the right-hand side of the rule, such as $A ::= \alpha \bullet \beta$, in which $|\alpha| = i$. When given a linked list node l and an environment e, a predicate π determines whether a production rule $A ::= \alpha$ can be applied to a non-terminal A, a mutator produces an updated environment e', and a selector yields a set of terminals that can replace a pre-terminal A. Therefore, some context-sensitive constraints can be defined with custom predicates, mutators and selectors.

3.2 Data Structures

To perform parsing in polynomial time and space, a GLL parser uses efficient data structures— descriptors, GSS nodes, and SPPF nodes— which we modify with linked lists, lookaheads and environments for constrained decoding. Linked lists, rather than integers, are used to represent parsing positions, to store terminal sequences that are dynamically extended during decoding. Lookaheads, which are predicted terminals that are not yet consumed by our GLL parser, are used to filter production rules that can be applied to non-terminals. Environments store information about context-sensitive constraints. Therefore, our GLL parser can track dynamically extended terminal sequences, filter applicable production rules with respect to predicted terminals, and keep context-sensitive information.

GLL parsing is performed by processing descriptors, which represent parsing states. A descriptor d is defined as a tuple (s, g, l, o, w, e) that has a slot s, a GSS non-terminal node g, a linked list node l, a lookahead o, an SPPF intermediate or symbol node w, and an environment e. In particular, the slot $s = A ::= \alpha \bullet \beta$ in the descriptor d means that α has been parsed and β is going to be parsed for the rule $A ::= \alpha\ \beta$.

We categorize pending descriptors, which can be processed but have not been processed yet, into three sets: $\mathcal{R}^{\mathrm{N}}$, $\mathcal{R}^{\mathrm{F}}$ and $\mathcal{R}^{\mathrm{L}}$. $\mathcal{R}^{\mathrm{N}}$ is the set of pending descriptors that have ε as lookaheads, $\mathcal{R}^{\mathrm{F}}$ is the set of pending descriptors that do not have ε as lookaheads and have non-last slots, and $\mathcal{R}^{\mathrm{L}}$ is the set of pending descriptors that do not have ε as lookaheads and have last slots, where a last slot s is like $A ::= \alpha \bullet \beta$ in which $\beta = \varepsilon$. With this categorization, we assign the lowest priorities to pending descriptors in $\mathcal{R}^{\mathrm{L}}$, so our GLL parser processes them last (Lines 24 to 26 in Algorithm B.2) and compute GSS transitive slot nodes (Section 3.3).

GSS nodes contain return information that is used when the current production rule is completely parsed by reaching the last slot of the production rule, and there are two types of GSS nodes: GSS non-terminal nodes and GSS slot nodes. A GSS non-terminal node g is defined as a tuple (A, l, o', e) that has a non-terminal A, a linked list node l, a lookahead o' and an environment e. A GSS slot node h is defined as a tuple (s, w, o) that has a slot s, an SPPF w and a lookahead o. The basic workflow is creating a GSS non-terminal node g' that is connected to the current GSS non-terminal node g via a GSS slot node h. When a production rule $A' ::= \alpha$ is applied to a non-terminal A' that exists in the right-hand side of a production rule $A ::= \beta\ A'\ \gamma$, then the non-terminal A' is stored in g' and the slot $A ::= \beta\ A' \bullet \gamma$, which indicates a return location, is stored in h.

In particular, our GSS slot node $h = (s, w, o)$ has a lookahead o that is stored not to be consumed when applying a production rule and that is later fetched to continue parsing. Once a lookahead is stored in a GSS slot node h, a new descriptor d can have ϵ as a lookahead. Having ϵ as a lookahead in a descriptor d means that an SPPF null node, which corresponds to a null parse forest, is being constructed until the current production rule is completely parsed.

SPPF nodes constitute a graph that compresses a parse forest for a string, and there are two types of SPPF nodes: SPPF intermediate or symbol nodes and SPPF packed nodes. An SPPF intermediate or symbol node w is defined as a tuple $\left(m, l^{\mathrm{L}}, l^{\mathrm{R}}, e\right)$ that has a slot or symbol m, a linked list node l^{L} that means a left extent, a linked list node l^{R} that means a right extent, and an environment e. An SPPF packed node v is a tuple (s, l) that has a slot s and a linked list node l that means a pivot. An SPPF

intermediate or symbol node w can have multiple child SPPF packed nodes, and an SPPF packed node v has two child SPPF intermediate or symbol nodes, w^{L} and w^{R}, that split w, which ranges from l^{L} to l^{R}, by a pivot l. In a binary splitting manner, SPPF nodes are constructed in polynomial time and space.

In addition, we added SPPF lazy nodes that defer constructing SPPF nodes until parsing is finished. An SPPF lazy node u is defined as a tuple (h^{τ}, g, w, e) that has a GSS transitive slot node h^{τ} (Section 3.3), a GSS non-terminal node g, an SPPF symbol node w and an environment e. By creating an SPPF lazy node u, our GLL parser does not repeat to construct SPPF symbol nodes for a right-recursive production rule. After parsing is finished, we can use an SPPF lazy node $u = (h^{\tau}, g, w, e)$ that is included in the final SPPF structure to construct SPPF symbol nodes by following edges from a GSS non-terminal node g to a GSS transitive slot node h^{τ}.

3.3 Right Recursion Optimization

Unlike ordinary GLL parsers, our GLL parser cannot use the FOLLOW sets to filter applicable production rules, because lookaheads should be chosen from FIRST sets, and the absence of FOLLOW sets makes our GLL parser inefficient for right-recursive production rules. When a right-recursive production rule is completely parsed, reductions are repeated, with constructing SPPF symbol nodes, until the first application of the right-recursive production rule is completed. The time cost of the recursive reductions is $O(|l|)$ in which $|l|$ is the length of a linked list that starts from l which is a part of the current descriptor $d = (s, g, l, o, w, e)$. Therefore, the time complexity of parsing the terminal sequence of l is at least $O(|l|^2)$ even if an $\mathrm{LL}(1)$ grammar is used.

For efficient reductions with right-recursive production rules, we adapt the right-recursion optimization method (Leo, 1991) into our GLL parser with introducing GSS transitive slot nodes. When there is a path in which each GSS non-terminal node has only one out-going edge to a GSS slot node, reduction from any of the GSS non-terminal nodes results in the same final return position. To prevent repeated reduction operations, a GSS transitive slot node directly provides the final return information for a reduction path.

For a GSS non-terminal node g to have a GSS transitive slot node h^{τ}, four conditions should be satisfied (Algorithm 1). First, g should not have ε as a lookahead (Line 3). Second, the number of out-going edges from g should be one (Line 7), where the number of out-going edges does not further increase assuming that pending descriptors in $\mathcal{R}^{\mathrm{L}}$ have the lowest priorities (Section 3.2). Third, the unique GSS slot node h that is connected from g should have a last slot (Line 10). Fourth, the slot s should not have a mutator $\mu = M(s)$ (Line 10); otherwise, updating an environment with μ can be skipped during reduction (Section 3.4).

Algorithm 1 Function to retrieve a GSS transitive slot node. g is a GSS non-terminal node, and $\bigcirc$ is the dummy GSS slot node, which is returned when the GSS non-terminal node g does not have a GSS transitive slot node.

1: **function** $\mathtt{trans}(g)$
2: $\quad (_, _, o, _) \leftarrow g$
3: $\quad$ **if** $o = \varepsilon$ **then**
4: $\quad\quad h^{\tau} \leftarrow \bigcirc$
5: $\quad$ **else if** $g \in \mathtt{domain}(\mathcal{T})$ **then**
6: $\quad\quad h^{\tau} \leftarrow \mathcal{T}(g)$
7: $\quad$ **else if** $|\mathtt{nodes}(g)| = 1$ **then**
8: $\quad\quad \{h\} \leftarrow \mathtt{nodes}(g)$
9: $\quad\quad (s, _, _) \leftarrow h$
10: $\quad\quad$ **if** $\mathtt{isLast}(s) \wedge s \notin \mathtt{domain}(M)$ **then**
11: $\quad\quad\quad \{g'\} \leftarrow \mathtt{nodes}(h)$
12: $\quad\quad\quad h' \leftarrow \mathtt{trans}(g')$
13: $\quad\quad\quad h^{\tau} \leftarrow \begin{cases} h', & \text{if } h' \neq \bigcirc \\ h, & \text{otherwise} \end{cases}$
14: $\quad\quad\quad \mathcal{T}(g) \leftarrow h^{\tau}$
15: $\quad\quad$ **else**
16: $\quad\quad\quad h^{\tau} \leftarrow \bigcirc$
17: $\quad$ **else**
18: $\quad\quad h^{\tau} \leftarrow \bigcirc$
19: $\quad$ **return** h^{τ}

3.4 GLL Parsing

Our GLL parser process descriptors with add, push and pop functions to construct SPPF nodes (Algorithm B.2). The add function (Algorithm B.3) makes a descriptor, then adds the descriptor to $\mathcal{R}^{\mathrm{N}}$, $\mathcal{R}^{\mathrm{F}}$ or $\mathcal{R}^{\mathrm{L}}$ as a pending descriptor, or adds the descriptor to $\mathcal{I}$ as an incomplete descriptor, which cannot be further processed until its lookahead is updated by a model (Section 3.5). The push function (Algorithm B.4) applies each valid production rule $A ::= \alpha$ for a non-terminal A, and creates a new GSS non-terminal node g' that is connected to the current GSS non-terminal node g via a GSS slot node h, then uses the add function to make

a descriptor that has the start slot $A ::= \bullet\, \alpha$ and g'. In particular, the push function does not create a duplicate GSS non-terminal node, so parsing is efficiently performed. The pop function (Algorithm B.9) performs reduction for each GSS slot node $h = (s, w', o)$ that is connected from the current GSS non-terminal node g, and combines the SPPF intermediate node w' with the current SPPF symbol node w into a new SPPF intermediate or symbol node w'', then uses the add function to make a descriptor that has w'' and the GSS non-terminal node g' that is connected from h.

In particular, our GSS parser uses predicates to conditionally apply production rules, and uses mutators to update environments when the parser reaches specific slots. When our GLL parser finds an applicable rule (Algorithm B.6), if a production rule r exists in the domain of the mapping Π, the predicate $\pi = \Pi(r)$ is used to compute the applicability $\upsilon = \pi(l, e)$; otherwise, r is always applicable (Algorithm B.7). When our GLL parser moves to a new slot (Line 21 in Algorithm B.2, Lines 11 and 20 in Algorithm B.5, Line 14 in Algorithm B.9), if the slot s exists in the domain of the mapping M, the mutator $\mu = M(s)$ is used to replace the current environment e with a new environment $e' = \mu(l, e)$; otherwise, e is kept (Algorithm B.8).

3.5 Constrained Decoding

The GLLD algorithm combines the beam search algorithm with our GLL parser to generate high-likelihood terminal sequences, which are stored in linked list nodes, that are valid with respect to a grammar, and to construct SPPF symbol nodes that correspond to the generated terminal sequences (Algorithm 2). For each time step during beam search, a model with a set θ of parameters computes a mapping ξ that assigns a linked list node l to a score vector $\xi(l)$ for next terminals (Lines 8 to 10). Concurrently, our GLL parser computes a mapping ϕ that assigns a linked list node l to a set $\phi(l)$ of FIRST sets (Line 11); in particular, while executing the getCandMap function, our GLL parser uses a selector $\sigma = \Sigma(A)$ for a pre-terminal A to compute a dynamically changing set $\sigma(l, e)$ of terminals, depending on a linked list node node l and an environment e, instead of $\mathsf{FIRST}(A)$ (Algorithm B.14). For a given $l \in B$, the score vector $\mathbf{c} = \xi(l)$ and the mask vector $\mathbf{m} = \mathtt{getUnifiedMask}(\phi(l), \mathcal{M})$ are added to a masked score vector $\mathbf{m}'$ (Lines 13 to 15), which is then used to collect top score linked

Algorithm 2 Function to perform beam search. θ is a parameter vector, e is an environment, x is and input text, κ is a beam size, η is the number of return sequences, T is the maximum number of iterations, and $\mathcal{M}$ is a mapping that assigns a set of terminals to a mask vector. An expression $\arg\max^{\kappa}_{\text{✧}\in\text{❖}} f(\text{✧})$ means choosing top-κ elements from a set ❖ with respect to a function f.

1: **function** beamSearch$(\theta, e, x, \kappa, \eta, T, \mathcal{M})$
2: $\quad$ initialize(e)
3: $\quad B \leftarrow \{(\text{nil}, 0)\}$
4: $\quad C \leftarrow \{\}$
5: $\quad t \leftarrow 0$
6: $\quad \upsilon \leftarrow \text{true}$
7: $\quad$ **while** $t < T \wedge \upsilon$ **do**
8: $\quad\quad \xi \leftarrow$ an empty mapping
9: $\quad\quad$ **for** $(l, c) \in B$ **do**
10: $\quad\quad\quad \xi(l) \leftarrow \mathtt{computeScores}(\theta, l, x, c)$
11: $\quad\quad \phi \leftarrow \mathtt{getCandMap}()$
12: $\quad\quad$ **for** $(l, _) \in B$ **do**
13: $\quad\quad\quad \mathbf{c} \leftarrow \xi(l)$
14: $\quad\quad\quad \mathbf{m} \leftarrow \mathtt{getUnifiedMask}(\phi(l), \mathcal{M})$
15: $\quad\quad\quad \mathbf{c}' \leftarrow \mathbf{c} + \mathbf{m}$
16: $\quad\quad\quad \boldsymbol{i} \leftarrow \arg\max^{(\kappa+\eta)}_{i \in 1,\ldots,|T|} \mathbf{c}'_i$
17: $\quad\quad\quad B' \leftarrow \{\}$
18: $\quad\quad\quad C' \leftarrow \{\}$
19: $\quad\quad\quad$ **for** $i \in \boldsymbol{i}$ **do**
20: $\quad\quad\quad\quad a \leftarrow \mathrm{ID}^{-1}(i)$
21: $\quad\quad\quad\quad$ **if** $a \neq \langle \mathtt{EOS} \rangle$ **then**
22: $\quad\quad\quad\quad\quad B' \leftarrow B' \cup \{(\mathtt{cons}(a, l), \mathbf{c}'_i)\}$
23: $\quad\quad\quad\quad$ **else**
24: $\quad\quad\quad\quad\quad C' \leftarrow C' \cup \{(\mathtt{cons}(a, l), \mathbf{c}'_i)\}$
25: $\quad\quad B \leftarrow \arg\max^{\kappa}_{(l,c) \in B'} c$
26: $\quad\quad C \leftarrow \arg\max^{\eta}_{(l,c) \in C \cup C'} c$
27: $\quad\quad \psi \leftarrow$ an empty mapping
28: $\quad\quad$ **for** $(l, _) \in B \cup (C \cap C')$ **do**
29: $\quad\quad\quad$ **if** $\mathtt{cdr}(l) \notin \mathtt{domain}(\psi)$ **then**
30: $\quad\quad\quad\quad \psi(\mathtt{cdr}(l)) \leftarrow \{\}$
31: $\quad\quad\quad \psi(\mathtt{cdr}(l)) \leftarrow \psi(\mathtt{cdr}(l)) \cup \{\mathtt{car}(l)\}$
32: $\quad\quad$ setLookaheads(ψ)
33: $\quad\quad t \leftarrow t + 1$
34: $\quad\quad \upsilon \leftarrow \min_{(l,c) \in C} c < \max_{(l,c) \in B} c$
35: $\quad$ **return** C

list node $\mathtt{cons}(a, l)$ in which a is a candidate of the next terminal (Lines 16 to 24). At the end of each time step, the beam B of incomplete hypotheses and the set C of completed hypotheses are updated (Lines 25 to 26), then the mapping ψ that assigns a linked list node l to a set of looka-

heads $\psi(l)$ is constructed (Lines 27 to 31) and sent to our GLL parser (Line 32). For a linked list node l from C, we can later get the set $\omega(l) = \{w \mid (g^{\text{root}}, w) \in \mathcal{P} \wedge \texttt{rightExtent}(w) = l\}$ of SPPF symbols nodes that corresponds to the linked list node l.

For efficient constrained decoding, we optimize the GLLD algorithm with multiprocessing and mask vector caching. GLLD uses one process for a model that produces terminals and multiple processes for GLL parsers that consume the terminals. When the GLL parser does not have any pending descriptor, it send the identifiers of sets of candidate terminals to the model (Algorithm B.12), then the model gets cached mask vectors (Nam and Lee, 2026) from the identifiers and unify them (Algorithm B.16). The unified mask vector is added to the score vector (Lines 14 to 15 in Algorithm 2), from which high-likelihood valid terminals can be acquired.

3.6 Complexities

We present the time complexities of our GLL parser with different grammars. To simplify the complexities, we assume that predicates and mutators are not used, and that the beam size κ and the number η of return sequences are one. In the following complexities, we use n as the length $|l|$ of the longest linked list node l during parsing. For an ε-free LL(1) grammar, the time complexity is $O(|\delta(\omega(l))|)$ in which $|\delta(\omega(l))|$ is the number of production rules that constitutes the parse tree $\delta(\omega(l))$ that is arbitrarily extracted from the set $\omega(l)$ of SPPF symbol nodes. For an LL(k) grammar, the time complexity is $O(|\zeta(P)|\, k^2 n)$, where $|\zeta(P)|$ is the number of all slots from the set P of production rules and k is the beam size. For a general CFG, the time complexity is $O(|\zeta(P)|\, n^3)$. When the beam size κ is more than one, the time complexities are multiplied by κ, resulting in $O(\kappa\, |\delta(\omega(l))|)$ for an LL(1) grammar, $O(\kappa\, |\zeta(P)|\, k^2 n)$ for an LL(k) grammar, and $O(\kappa\, |\zeta(P)|\, n^3)$ for a general CFG. Compared to the Earley parser, the time complexities of our GLL parser for an LL(k) grammar and a general CFG are the same as those of the Earley parser (Leo, 1991; Opedal et al., 2023). However, our GLL parser is more efficient than the Earley parser when a grammar is exactly or nearly ε-free LL(1), as our GLL parser apply production rules that are filtered with respect to lookaheads (Line 12 in Algorithm B.6).

Algorithm 3 Function to perform semantic parsing.

```
function glldSP(θ, e, x, κ, T, M)
    η ← 1
    {(l, _)} ← beamSearch(θ, e, x, κ, η, T, M)
    return ρ(δ(ω(l)))
```

3.7 Semantic Parsing

We use the GLLD algorithm for the semantic parsing task which maps an utterance x to a logical form z (Warren and Pereira, 1982) which is evaluated on a knowledge $\mathcal{K}$ to produce a denotation $[\![z]\!]_{\mathcal{K}}$ (Liang et al., 2011). Our semantic parsing function `glldSP` uses the `beamSearch` function to produce a logical form $\rho(\delta(\omega(l)))$ which is synthesized from a parse tree $\delta(\omega(l))$ in which l is a high-likelihood linked list node (Algorithm 3). When given a parse tree $\triangle$, the ρ function constructs a logical form $\rho(\triangle)$ in a bottom-up manner by applying logical form templates (Nam and Lee, 2026) that are associated with production rules. For deterministic mapping from an utterance x to a logical form $\rho(\delta(\omega(l)))$, we use a non-ambiguous grammar or a grammar that enables the ρ function to produce the same logical form from an arbitrary parse tree $\delta(\omega(l))$ for a linked list node l. For simplicity, we define a parameterized semantic parser f_θ as follows.

$$f_\theta(x) = \texttt{glldSP}(\theta, e, x, \kappa, T, \mathcal{M}). \qquad (1)$$

Rather than directly generating an executable logical form, our model generates a simplified representation that is a sequence of operators and arguments, and the representation is stored in the output linked list node l as terminals (Appendix D). The simplified representation reduces the length of predicted terminals, and enhances decoding speed for an input utterance x.

The goal of our semantic parser f_θ is to predict a correct logical form $f_\theta(x)$ whose denotation $[\![f_\theta(x)]\!]_{\mathcal{K}}$ is identical to the gold denotation y. To train our semantic parser f_θ, we use strong supervision from gold logical forms and weak supervision from gold denotations. For strongly-supervised learning, we maximize the log-likelihood $\log p_\theta(l \mid x)$ in which l is the linked list node that corresponds to the gold logical form z. For weakly-supervised learning, we maximize the log-likelihood $\log p_\theta(l \mid x)$, in which l is the highest-likelihood linked list node, which is found by the `beamSearch` function, that satisfies

$[\![\rho(\delta(\omega(l)))]\!]_{\mathcal{K}} = y$ for the gold denotation y (Min et al., 2019).

4 Experiments

We experimented on our semantic parser with the NLVR benchmark (Suhr et al., 2017), whose task is to predict a boolean value that indicates whether an utterance x is correct with respect to a knowledge base $\mathcal{K}$ that contain information of objects. To tackle the task, our semantic parser maps an utterance x to a logical form $f_\theta(x)$ which is then evaluated to produce a denotation $[\![f_\theta(x)]\!]_{\mathcal{K}}$ that is a boolean value. The implementation details are provided in in Appendix C

4.1 Setup

We use a logical form language (Goldman et al., 2018) that composes functions with typed variables, for which we define context-sensitive constraints. The logical form language uses lambda variables, such as `x` or `y`, whose types are determined by the position of lambda functions, such as `(λ (x) (and ···))`. For valid uses of lambda variables, a newly created lambda variable is stored in an environment and categorized by its type, and the lambda variable is later removed from the environment when the scope of the corresponding lambda function ends. Using the environment, parser can provides available variables for a specific type. We also define other context-sensitive constraints, such as restricting the maximum number of lambda functions or preventing successive use of `and`, such as `(and (and (and ···)))`.

To demonstrate the effectiveness of constraints, we apply different decoding strategies: (1) unconstrained decoding, (2) constrained decoding with a CFG G^{F}, and (3) constrained decoding with a grammar G^{S} that has predicates, mutators and selectors. Following Goldman et al. (2018), when our semantic parser fails to find a logical form, especially when decoding without any constraints or context-sensitive constraints, we treat the majority denotation label `true` as the output denotation.

We evaluate our semantic parser with two measures: accuracy and consistency (Goldman et al., 2018). Accuracy is the fraction of examples where the denotation of an example is correctly predicted. Consistency is the fraction of groups (Appendix C.1) where all denotations in a group are correctly predicted. Consistency is a more reliable measure than accuracy, so we set consistency as the first criterion for early stopping and for selecting the best model instances; if a tie occurs, we use accuracy as the second criterion. The beam size κ for evaluation is one by default, and we additionally change κ to four in Table 1. All reported accuracies and consistencies are evaluated in a single-run setting

4.2 Results

We list accuracies and consistencies of previous semantic parsers and ours (Table 1). Our weakly-supervised semantic parsers, which were based on BART-base (Lewis et al., 2020), had higher accuracies and consistencies, compared to the previous semantic parsers which were based on LSTMs (Hochreiter and Schmidhuber, 1997). In particular, our proposed semantic parsers achieved substantial improvements in consistency over previous semantic parsers.

We also compare accuracies and consistencies of our models with different decoding strategies. On the test set, our semantic parsers with GLLD achieved higher consistencies in the public or hidden test set. In particular, the weakly-supervised semantic parser with GLLD and G^{S} achieved highest accuracies and consistencies, and the semantic parser further increased accuracies and consistencies in the hidden test set with the beam size four. Therefore, the search steps during weakly-supervised learning benefited from context-sensitive constraints to find a linked list node l that satisfies $[\![\rho(\delta(\omega(l)))]\!]_{\mathcal{K}} = y$ for a gold denotation y, as the search space is reduced by the constraints.

To measure efficiency of the GLLD algorithm, we computed the average time to generate a token during decoding. The time was calculated by dividing total decoding time by the sum of lengths of all output sequences in batches. The experiment was performed on a computer with an AMD EPYC 7502 32-Core Processor, in which we used five physical cores, and on a NVIDIA RTX A5000. We used one process for BART-base and four processes for GLL parsers.

We report the average time to generate a token during decoding (Table 2). When the beam size was one and the batch size was one, decoding without GLLD and with GLLD did not have a big gap of time. This result demonstrates the efficiency of GLLD. However, the gap widened when the beam size or the batch size increased, although parsers concurrently process descriptors. The reason of the increasing gap is the inevitable operations on

Table 1: Accuracies and consistencies on the NLVR benchmark. P-Test means the public test set, and H-Test means the hidden test set. Semantic parsers with the superscript letter ✧ are trained only with abstract examples.

Model	Dev.		P-Test		H-Test	
	Acc.	Con.	Acc.	Con.	Acc.	Con.
Goldman et al. (2018)✧	77.7	52.4	76.6	51.8	-	-
Goldman et al. (2018)	85.7	67.4	84.0	65.0	82.5	63.9
Dasigi et al. (2019)	85.4	64.8	82.4	61.3	82.9	64.3
Gupta et al. (2021)	89.6	75.9	86.3	71.0	89.5	74.0
Lee et al. (2023)	90.5	78.8	89.4	74.2	89.4	76.3
Ours✧ w/o constraints	73.4	44.6	73.1	45.9	73.3	44.7
Ours✧ w/ GLLD & G^{F}	71.1	47.6	71.0	46.6	71.7	46.6
Ours✧ w/ GLLD & G^{S}	71.1	47.6	71.1	46.6	71.9	47.0
Ours w/o constraints	94.0	85.0	89.3	77.1	90.4	79.3
Ours w/ GLLD & G^{F}	92.6	83.5	89.8	78.2	90.3	79.3
Ours w/ GLLD & G^{S}	**94.4**	**87.3**	**90.8**	**80.8**	91.8	82.0
& beam size = 4	94.1	86.9	90.3	**80.8**	**91.9**	**82.3**

Table 2: Average time to generate a token during decoding. The average time is computed over the public test set, whose examples are grouped, that is duplicated 40 times.

Model	Time (μs)			
	Batch size = 1		Batch size = 16	
	Beam size = 1	Beam size = 4	Beam size = 1	Beam size = 4
Ours w/o constraints	4297	6592	705	1439
Ours w/ GLLD & G^{S}	4755	7990	1064	3389

CPUs, such as communication between the model and the parsers. This performance gap would be ignored when we use a larger model that is slower than BART-base.

5 Related Work

Poesia et al. (2022) and Li et al. (2025) proposed constrained decoding frameworks that handle context-sensitive constraints by representing the next strings as regular expressions. The frameworks enable incorporating specific completion engine or parser for context-sensitive constraints. In contrast, we adapt formal methods (Afroozeh and Izmaylova, 2015b; Parr and Fisher, 2011) for context-sensitive constraints into our GLL parser.

The token misalignment problem occurs as terminals in grammar do not correspond to vocabulary of an autoregressive model, and many previous methods address this problem (Poesia et al., 2022; Beurer-Kellner et al., 2024; Dong et al., 2025; Li et al., 2025; Dong et al., 2025). In contrast, we do not address the token misalignment problem, as we treat tokens as terminals. However, to incorporate context-sensitive constraints, GLLD with a less tight CFG can additionally support another constrained decoding method that address token misalignment. In addition, as our contributions are independent to the previous methods, combining GLLD with the methods, such as data-dependent token checking (Dong et al., 2025; Li et al., 2026), is a promising direction for future research.

6 Conclusion

We proposed the GLLD algorithm that uses efficient GLL parsers for decoding with context-free and context-sensitive constraints. We augmented a CFG with predicates, mutators and selectors for context-sensitive constraints. We modified the data structures of GLL parsers with linked list nodes, lookaheads and environments for constrained decoding. We optimized right recursion in our GLL parser by prioritizing descriptors. The GLLD algorithm finds high-likelihood terminal sequences that are valid with respect to our grammar in polynomial time. We applied the GLLD algorithm to the semantic parsing task, and GLLD improved accuracy and consistency on the NLVR benchmark. Finally, GLLD achieved fast decoding speed.

7 Limitations

The GLLD algorithm does not address token misalignment problem (Poesia et al., 2022; Beurer-Kellner et al., 2024), as the algorithm assumes the vocabulary of a model is identical to the set T of terminals. Therefore, GLLD can reduce the likelihood of the output text that has complex syntax.

In addition, our experiments are limited. First, we experimented only with the NLVR benchmark, so effectiveness of GLLD on other benchmarks that need context-sensitive constraints is uncertain. Second, we used only the BART-base model; therefore, effectiveness in other LLMs, which is especially larger than BART-base, is uncertain.

References

Ali Afroozeh and Anastasia Izmaylova. 2015a. Faster, practical GLL parsing. In *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9031 of *Lecture Notes in Computer Science*, pages 89–108. Springer.

Ali Afroozeh and Anastasia Izmaylova. 2015b. One parser to rule them all. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 151–170. ACM.

John Aycock and R. Nigel Horspool. 2002. Practical earley parsing. *Comput. J.*, 45(6):620–630.

Artur Baranowski and Nico Hochgeschwender. 2021. Grammar-constrained neural semantic parsing with LR parsers. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, volume ACL/IJCNLP 2021 of *Findings of ACL*, pages 1275–1279. Association for Computational Linguistics.

Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2024. Guiding llms the right way: Fast, non-invasive constrained generation. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. 2021. Autoregressive entity retrieval. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Shuang Chen, Qian Liu, Zhiwei Yu, Chin-Yew Lin, Jian-Guang Lou, and Feng Jiang. 2021. Retrack: A flexible and efficient framework for knowledge base question answering. In *Proceedings of the Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL 2021 - System Demonstrations, Online, August 1-6, 2021*, pages 325–336. Association for Computational Linguistics.

Pradeep Dasigi, Matt Gardner, Shikhar Murty, Luke Zettlemoyer, and Eduard H. Hovy. 2019. Iterative search for weakly supervised semantic parsing. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 2669–2680. Association for Computational Linguistics.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ziyi Xu, Yilong Zhao, Ruihang Lai, and Tianqi Chen. 2025. Xgrammar: Flexible and efficient structured generation engine for large language models. In *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*. OpenReview.net/mlsys.org.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

Omer Goldman, Veronica Latcinnik, Ehud Nave, Amir Globerson, and Jonathan Berant. 2018. Weakly supervised semantic parsing with abstract examples. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 1809–1819. Association for Computational Linguistics.

Nitish Gupta, Sameer Singh, and Matt Gardner. 2021. Enforcing consistency in weakly supervised semantic parsing. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 2: Short Papers), Virtual Event, August 1-6, 2021*, pages 168–174. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 417–430. ACM.

Donald E. Knuth. 1965. On the translation of languages from left to right. *Inf. Control.*, 8(6):607–639.

Terry Koo, Frederick Liu, and Luheng He. 2024. Automata-based constraints for language model decoding. In *First Conference on Language Modeling*.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 1516–1526. Association for Computational Linguistics.

Jaebok Lee and Hyeonjeong Shin. 2025. SPARKLE: enhancing SPARQL generation with direct KG integration in decoding. *Expert Syst. Appl.*, 289:128263.

Kang-il Lee, Segwang Kim, and Kyomin Jung. 2023. Weakly supervised semantic parsing with execution-based spurious program filtering. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 6884–6894. Association for Computational Linguistics.

Joop M. I. M. Leo. 1991. A general context-free parsing algorithm running in linear time on every LR (k) grammar without using lookahead. *Theor. Comput. Sci.*, 82(1):165–176.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. Association for Computational Linguistics.

Lingxiao Li, Yiwei Zhao, and 1 others. 2025. Correctness-guaranteed code generation via constrained decoding. In *Second Conference on Language Modeling*.

Linzhang Li, Yixin Dong, Guanjie Wang, Ziyi Xu, Alexander Jiang, and Tianqi Chen. 2026. Xgrammar 2: Dynamic and efficient structured generation engine for agentic llms. *CoRR*, abs/2601.04426.

Percy Liang, Michael I. Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, pages 590–599. The Association for Computer Linguistics.

Sewon Min, Danqi Chen, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2019. A discrete hard EM approach for weakly supervised question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 2851–2864. Association for Computational Linguistics.

Michal Moskal. 2024. llguidance. `https://pypi.org/project/llguidance/`.

Daehwan Nam and Gary Geunbae Lee. 2026. Semantic parsing with candidate expressions for knowledge base question answering. *Expert Syst. Appl.*, 306:130564.

Anthony G Oettinger. 1961. *Automatic syntactic analysis and the pushdown store*. American Mathematical Society.

Andreas Opedal, Ran Zmigrod, Tim Vieira, Ryan Cotterell, and Jason Eisner. 2023. Efficient semiring-weighted earley parsing. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 3687–3713. Association for Computational Linguistics.

Terence Parr and Kathleen Fisher. 2011. Ll(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 425–436. ACM.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

Elizabeth Scott and Adrian Johnstone. 2009. GLL parsing. In *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications, LDTA 2009, York, UK, March 27-28, 2009*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 177–189. Elsevier.

Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Sci. Comput. Program.*, 78(10):1828–1844.

Richard Shin, Christopher H. Lin, Sam Thomson, Charles Chen, Subhro Roy, Emmanouil Antonios Platanios, Adam Pauls, Dan Klein, Jason Eisner, and Benjamin Van Durme. 2021. Constrained language models yield few-shot semantic parsers. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 7699–7715. Association for Computational Linguistics.

Alane Suhr, Mike Lewis, James Yeh, and Yoav Artzi. 2017. A corpus of natural language for visual reasoning. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*, pages 217–223. Association for Computational Linguistics.

Gabriele Tuccio, Luana Bulla, Maria Madonia, Aldo Gangemi, and Misael Mongiovì. 2025. GRAMMAR-LLM: grammar-constrained natural language generation. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July*

27 - August 1, 2025, volume ACL 2025 of *Findings of ACL*, pages 3412–3422. Association for Computational Linguistics.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2025. Syncode: LLM generation with grammar augmentation. *Trans. Mach. Learn. Res.*, 2025.

David H. D. Warren and Fernando C. N. Pereira. 1982. An efficient easily adaptable system for interpreting natural language queries. *Am. J. Comput. Linguistics*, 8(3-4):110–122.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, 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, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP 2020 - Demos, Online, November 16-20, 2020*, pages 38–45. Association for Computational Linguistics.

Shan Wu, Bo Chen, Chunlei Xin, Xianpei Han, Le Sun, Weipeng Zhang, Jiansong Chen, Fan Yang, and Xunliang Cai. 2021. From paraphrasing to semantic parsing: Unsupervised semantic parsing via synchronous semantic decoding. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 5110–5121. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 440–450. Association for Computational Linguistics.

A Notation

We summarize grammar components in Table A.1, global variables in Table A.2, local variables in Table A.3, constants in Table A.4, and utility functions in Tables A.5 to A.7.

B Pseudo-Code

We provide full pseudo-code in Algorithms 1, 2 and B.1 to B.17.

C Implementation Details

C.1 Benchmark

Examples in the dataset of the NLVR benchmark are grouped and each group has mostly four examples that share the same utterance but that have

Table A.1: Grammar components and special sets.

Symbol	Description
Grammar components	
G	The grammar.
N	The set of non-terminals.
N^{G}	The set of non-terminals that are not pre-terminals.
N^{P}	The set of pre-terminals.
T	The set of terminals.
P	The set of production rules.
S	The start non-terminal.
S'	The augmented start non-terminal.
Π	The mapping that assigns a rule to a predicate.
M	The mapping that assigns a slot to a mutator.
Σ	The mapping that assigns a pre-terminal to a selector.
Special sets	
L	The set of linked list nodes.
E	The set of environments.
$\zeta(P)$	The set of slots of rules in P.
$\zeta(r)$	The set of slots of a rule r.
$\mathbb{R}$	The set of real numbers.

Algorithm B.1 Function to initialize a GLL parser.

function `initialize`(e)
  $\mathcal{U} \leftarrow \{\}$
  $\mathcal{R}^{\mathrm{N}} \leftarrow \{\}$
  $\mathcal{R}^{\mathrm{F}} \leftarrow \{\}$
  $\mathcal{R}^{\mathrm{L}} \leftarrow \{\}$
  $\mathcal{I} \leftarrow \{\}$
  $\mathcal{G} \leftarrow \{\}$
  $\mathcal{P} \leftarrow \{\}$
  $\mathcal{T} \leftarrow$ an empty mapping
  $\mathcal{L} \leftarrow \{\}$
  $l \leftarrow$ nil
  $o \leftarrow \square$
  $r^{\mathrm{start}} \leftarrow S' ::= \langle \texttt{BOS} \rangle\ S\ \langle \texttt{EOS} \rangle$
  $g^{\mathrm{root}} \leftarrow (\texttt{lhs}(r^{\mathrm{start}}), l, o, e)$
  $\mathcal{G} \leftarrow \mathcal{G} \cup \{g^{\mathrm{root}}\}$
  $s \leftarrow \texttt{slot}(r^{\mathrm{start}})$
  $e' \leftarrow \texttt{update}(s, l, e)$
  $w \leftarrow \texttt{getNodeI}(s, l, e')$
  $\texttt{add}(s, g^{\mathrm{root}}, l, o, w, e')$
  $\texttt{process}()$

Table A.2: Global variables.

Symbol	Description
$\mathcal{U}$	The set of all descriptors
$\mathcal{R}^{\mathrm{N}}$	The set of pending descriptors that have ε as lookaheads.
$\mathcal{R}^{\mathrm{F}}$	The set of pending descriptors that do not have ε as lookaheads and have non-last slots.
$\mathcal{R}^{\mathrm{L}}$	The set of pending descriptors that do not have ε as lookaheads and have last slots.
$\mathcal{I}$	The set of incomplete descriptors that cannot be further processed until lookaheads are updated.
$\mathcal{G}$	The set of GSS non-terminal nodes.
$\mathcal{P}$	The set of pairs of popped GSS non-terminal nodes and SPPF nodes.
g^{root}	The GSS root non-terminal node.
$\mathcal{T}$	The mapping that assigns a GSS non-terminal node to a GSS transitive slot node.
$\mathcal{L}$	The set of pairs of SPPF symbol nodes and SPPF lazy nodes.

Table A.3: Local variables.

Symbol	Description
$_$	An unused variable.
α, β, γ	A string.
b	A symbol.
A	A non-terminal.
a	A terminal.
r	A rule.
s	A slot.
m	A slot or symbol.
l	A linked list node (a, l') or nil.
d	A descriptor (s, g, l, o, w, e).
g	A GSS non-terminal node (A, l, o, e).
h	A GSS slot node (s, w, o).
h^{τ}	A GSS transitive slot node.
w	An SPPF intermediate or symbol node $\big(m, l^{\mathrm{L}}, l^{\mathrm{R}}, e\big)$.
v	An SPPF packed node (s, l).
u	An SPPF lazy node (h^{τ}, g, w, e).
o	A lookahead.
e	An environment.
π	A predicate.
μ	A mutator.
σ	A selector.
ϕ	A mapping that assigns a linked list node to a set of FIRST sets.
ψ	A mapping that assigns a linked list node to a set of lookaheads.
ξ	A mapping that assigns a linked list node to a score vector.
$\boldsymbol{a}$	A set of terminals.
$\widehat{\boldsymbol{a}}$	A set of sets of terminals.
$\boldsymbol{r}$	A set of rules.
i	An index.
$\boldsymbol{i}$	A set of indices.
c	A score.
$\mathbf{c}$	A score vector.
$\mathbf{m}$	A mask vector.
υ	A boolean value.
$\mathcal{M}$	A mapping that assigns a set of terminals to a mask vector.
$\boldsymbol{w}$	A set of SPPF intermediate or symbol nodes.

different knowledge bases. The training, the validation, the public test and the hidden test splits respectively have 12410, 989, 990 and 985 examples, and 3163, 267, 266 and 266 groups. The dataset of the NLVR benchmark is under CC-BY-4.0.

C.2 Model

We use BART-base (Lewis et al., 2020) as our model to computes the scores for next tokens in `beamSearch`. BART-base consists of approximately 139 million parameters, and is distributed under the open-source MIT License.

C.3 Decoding

We use the implementation of the beam search algorithm in the transformers library (Wolf et al., 2020). To incorporate constraints into beam search, we add mask vectors to score vectors by using the `LogitsProcessor` API.

C.4 Training

In strongly-supervised learning, we sample newly-created concrete examples from 106 abstract examples for every batch, and our model, which is BART-base (Lewis et al., 2020), learns from the concrete examples. For each batch during strongly-supervised learning, our model is evaluated with different decoding strategies, and we choose the best checkpoint of parameters for each decoding strategy. The number of epochs is 25, and the batch

Table A.4: Constants.

Symbol	Description
nil	The end of a linked list.
$\square$	The dummy lookahead.
$\triangle$	The dummy SPPF.
$\bigcirc$	The dummy GSS slot node.
ε	The empty string.
$\langle$BOS$\rangle$	The beginning-of-sequence symbol.
$\langle$EOS$\rangle$	The end-of-sequence symbol.

Table A.5: First part of utility functions.

Symbol	Description
car(l)	When given a linked list node $l = (a, l')$, it returns the value a.
cdr(l)	When given a linked list node $l = (a, l')$, it returns the other linked list node l'.
cons(o, l)	When given a value a and a linked list node l, it constructs and returns a new linked list node (a, l).
lhs(r)	When given a rule $r = A ::= \alpha$, it returns the left-hand side A.
rhs(r)	When given a rule $r = A ::= \alpha$, it returns the right-hand side α.
slot(r)	When given a rule $r = A ::= \alpha$, it returns the first slot $A ::= \bullet\, \alpha$.
rule(s)	When given a slot $s = A ::= \alpha \bullet \beta$, it returns the rule $A ::= \alpha\, \beta$.
left(s)	When given a slot $s = A ::= \alpha\, b \bullet \beta$, it returns the left symbol b.
right(s)	When given a slot $s = A ::= \alpha \bullet b\, \beta$, it returns the right symbol b.
shift(s)	When given a slot $s = A ::= \alpha \bullet b\, \beta$, it returns the next slot $A ::= \alpha\, b \bullet \beta$.
isLast(s)	When given a slot $s = A ::= \alpha \bullet \beta$, it returns true if $\beta = \varepsilon$; otherwise, it returns false.

Table A.6: Second part of utility functions

Symbol	Description
leftExtent(w)	When given a SPPF intermediate or symbol node $w = (m, l^{\mathrm{L}}, l^{\mathrm{R}}, e)$, it returns the linked list node l^{L} which means the left extent.
rightExtent(w)	When given a SPPF intermediate or symbol node $w = (m, l^{\mathrm{L}}, l^{\mathrm{R}}, e)$, it returns the linked list node l^{R} which means the right extent.
isNullNode(w)	When given a SPPF intermediate or symbol node $w = (m, l^{\mathrm{L}}, l^{\mathrm{R}}, e)$, it returns true if $l^{\mathrm{L}} = l^{\mathrm{R}}$; otherwise, it returns false.
leftChild(v)	When given a SPPF packed node v, it returns the left child of v.
rightChild(v)	When given a SPPF packed node v, it returns the right child of v.
sppf$(m, l^{\mathrm{L}}, l^{\mathrm{R}}, e)$	It makes and returns an SPPF node $w = (m, l^{\mathrm{L}}, l^{\mathrm{R}}, e)$ if w has not been created; otherwise, it returns the existing SPPF node w.
sppfL(h^{τ}, g, w, e)	It makes and returns an SPPF lazy node $u = (h^{\tau}, g, w, e)$ if u has not been created; otherwise, it returns the existing SPPF lazy node u.

size is 16. We use an NVIDIA RTX 2080 Ti GPU in strongly-supervised learning, and the training time is roughly 17 minutes.

In weakly-supervised learning, we fine-tune the strongly-supervised model with a decoding strategy from 3163 groups of the training split of NLVR. In search steps of weakly-supervised learning (Nam and Lee, 2026), we use the beamSearch function with the decoding strategy to find the high-likelihood linked list node l for each group, where the condition $[\![\rho(\delta(\omega(l)))]\!]_{\mathcal{K}} = y$ is satisfied for each pair of knowledge base $\mathcal{K}$ and y in a example of the group. The beam size κ and the number η of return sequences are eight in the search steps. In each optimization step, a model is trained for 8 epochs where the batch size is 16. We use four

Algorithm B.2 Function to process pending descriptors.

```
function process()
  while (R^N ≠ {}) ∨ (R^F ≠ {}) ∨ (R^L ≠ {}) do
    if R^N ≠ {} then
      (s, g, l, o, w, e) ← extract(R^N)
      if isLast(s) then
        pop(g, l, w, e)
      else
        push(shift(s), g, l, o, w, e)
    else if R^F ≠ {} then
      (s, g, l, o, w, e) ← extract(R^F)
      b ← right(s)
      s' ← shift(s)
      if b ∈ N^G then
        push(s', g, l, o, w, e)
      else
        l' ← cons(o, l,)
        if b ∈ N^P then
          w^R ← getNodePT(b, o, l', e)
        else
          w^R ← getNodeT(o, l', e)
        e' ← update(s', l', e)
        w' ← getNodeF(s', w, w^R, e')
        add(s', g, l', □, w', e')
    else
      (s, g, l, o, w, e) ← extract(R^L)
      pop(g, l, w, e)
```

Algorithm B.3 Function to add a descriptor to a set of descriptors.

```
function add(s, g, l, o, w, e)
  d ← (s, g, l, o, w, e)
  if d ∉ U then
    U ← U ∪ {d}
    if o = ε then
      R^N ← R^N ∪ {d}
    else if isLast(s) then
      if ¬isNullNode(w) then
        R^L ← R^L ∪ {d}
    else if o = □ then
      b ← right(s)
      if b ∈ N^G then
        if isDynamic(b) then
          R^F ← R^F ∪ {d}
        else
          if isNullable(b) then
            R^F ← R^F ∪ {d}
          I ← I ∪ {d}
      else
        I ← I ∪ {d}
    else
      R^F ← R^F ∪ {d}
```

NVIDIA A100 Tensor Core (80GB) GPUs, with which we use data parallelism especially to reduce the time of search steps, and the training time is roughly 80 minutes.

D Examples of Semantic Parsing

We attach examples of semantic parsing in Table D.1. The examples from the public split. For each example, a linked linked node l, which is represented as the terminal sequence it stores, and a logical form $\rho(\delta(\omega(l)))$ are predicted by a weakly-supervised semantic parser with GLLD and G^{S}.

E Use of AI Assistants

To improve the writing quality of this paper, we used AI assistants, which suggested linguistic expressions, corrected grammatical errors, and proofread the manuscript.

Algorithm B.4 Function to push a GSS non-terminal node.

```
function push(s, g, l, o, w, e)
  A ← left(s)
  if o ≠ ε ∧ (o ≠ □ ∨ isDynamic(A)) then
    create(s, g, l, □, o, w, e)
  if ((¬isLast(s)) ∨ o ∈ {□, ε}) ∧ isNullable(A) then
    create(s, g, l, o, ε, w, e)
```

Table A.7: Third part of utility functions

Symbol	Description
$\texttt{extract}(\mathcal{R})$	It extracts an element from the set $\mathcal{R}$, and return the element.
$\texttt{connect}(\star, \diamond)$	It adds a directed edge from a node $\star$ to another node $\diamond$.
$\texttt{nodes}(\star)$	It returns the set of nodes that are connected from a node $\star$, where the connections are made by the `connect` function.
$\texttt{isNullable}(\alpha)$	It returns true if there is a derivation $\alpha \Rightarrow^* \varepsilon$; otherwise, it returns false.
$\texttt{isDynamic}(A)$	It returns true if the set $\mathsf{FIRST}(A)$ dynamically changes depending on a linked list node l and an environment e. Formally, $\texttt{isDynamic}(A) = \text{true}$ when there is a leftmost derivation $A \overset{r_1}{\Longrightarrow} \alpha_1 \overset{r_2}{\Longrightarrow} \alpha_2 \ldots \overset{r_n}{\Longrightarrow} \alpha_n$ where $\forall i \in \{1, \ldots, n-1\}$, $\alpha_i = b_i \beta_i$ and $b_i \notin T \cup N^{\mathrm{P}}$, and either (1) $\alpha_n = a\beta$ and $a \in T$, and $\exists i \in \{1, \ldots, n-1\}$ s.t. $r_i \in \texttt{domain}(\Pi)$, or (2) $\alpha_n = A'\beta$, $A' \in N^{\mathrm{P}}$ and $A' \in \texttt{domain}(\Sigma)$
$\texttt{domain}(f)$	It returns the domain of a mapping f.
$\mathrm{ID}(a)$	It returns the identifier i of a terminal a, where i is used as an index of a score vector or a mask vector.
$\omega(l)$	It returns the set $\boldsymbol{w} = \{w \mid (g^{\mathrm{root}}, w) \in \mathcal{P} \wedge \texttt{rightExtent}(w) = l\}$ of SPPF symbols nodes that corresponds to a linked list node l.
$\delta(\boldsymbol{w})$	It returns an arbitrary parse tree that is extracted from a set of SPPF symbol nodes $\boldsymbol{w}$.
$\rho(\triangle)$	It returns a logical form that is transformed from a parse tree $\triangle$.

Algorithm B.5 Function to create a GSS non-terminal node.

```
function create(s, g, l, o, o', w, e)
    A ← left(s)
    g' ← (A, l, o', e)
    if g' ∉ 𝒢 then
        𝒢 ← 𝒢 ∪ {g'}
        h ← (s, w, o)
        connect(g', h)
        connect(h, g)
        for r ∈ alter(A, l, o', e) do
            s' ← slot(r)
            e' ← update(s', l, e)
            w' ← getNodeI(s', l, e')
            add(s', g', l, o', w', e')
    else
        h ← (s, w, o)
        connect(g', h)
        connect(h, g)
        for w' s.t. (g', w') ∈ 𝒫 do
            (_, _, l^R, e') ← w'
            e'' ← update(s, l^R, e')
            w'' ← getNodeF(s, w, w', e'')
            add(s, g, l^R, o, w'', e'')
```

Algorithm B.6 Function to find a set of applicable alternatives of a non-terminal.

```
function alter(A, l, o, e)
    r ← {}
    if o = ε then
        for r ∈ P s.t. lhs(r) = A ∧ isNullable(rhs(r)) do
            if isApplicable(r, l, e) then
                r ← r ∪ {r}
    else if o = □ then
        for r ∈ P s.t. lhs(r) = A ∧ rhs(r) ≠ ε do
            if isApplicable(r, l, e) then
                r ← r ∪ {r}
    else
        for r ∈ P s.t. lhs(r) = A ∧ o ∈ FIRST(rhs(r)) do
            r ← r ∪ {r}
    return r
```

Algorithm B.7 Function to check whether a production rule is applicable.

```
function isApplicable(r, l, e)
    if ∃r ∈ domain(Π) then
        π = Π(r)
        υ = π(l, e)
    else
        υ = true
    return υ
```

Algorithm B.8 Function to update an environment.

```
function update(s, l, e)
    if ∃s ∈ domain(M) then
        μ = M(s)
        e′ = μ(l, e)
    else
        e′ = e
    return e′
```

Algorithm B.9 Function to pop a GSS non-terminal node.

```
function pop(g, l, w, e)
    if (g, w) ∉ 𝒫 then
        𝒫 ← 𝒫 ∪ {(g, w)}
        h^τ ← trans(g)
        if h^τ ≠ ○ then
            (s, _, o) ← h^τ
            {g′} ← nodes(h^τ)
            w′ ← getNodeL(h^τ, g, w, e)
            add(s, g′, l, o, w′, e)
        else
            for h ∈ nodes(g) do
                (s, w′, o) ← h
                {g′} ← nodes(h)
                e′ ← update(s, l, e)
                w″ ← getNodeF(s, w′, w, e′)
                add(s, g′, l, o, w″, e′)
```

Algorithm B.10 Functions to construct SPPF nodes. `getNodeI` is for an SPPF initial intermediate node or an SPPF null node. `getNodePT` is for an SPPF pre-terminal node. `getNodeF` is for an SPPF intermediate or symbol node. `getNodeL` is for an SPPF lazy node.

```
function getNodeI(s, l, e)
    m ← { lhs(rule(s)),  if isLast(s)
        { s,             otherwise
    w ← sppf(m, l, l, e)
    if isLast(s) then
        pack(w, s, △, l, △)
    return w

function getNodeT(a, l^R, e)
    return sppf(a, cdr(l^R), l^R, e)

function getNodePT(A, a, l^R, e)
    l^L ← cdr(l^R)
    w ← sppf(A, l^L, l^R, e)
    s ← slot(A ::= a)
    w^L ← sppf(s, l^L, l^L, e)
    w^R ← getNodeT(a, l^R, e)
    pack(w, shift(s), w^L, l^L, w^R)
    return w

function getNodeF(s, w^L, w^R, e)
    m ← { lhs(rule(s)),  if isLast(s)
        { s,             otherwise
    l^L ← leftExtent(w^L)
    l^R ← rightExtent(w^R)
    w ← sppf(m, l^L, l^R, e)
    l ← rightExtent(w^L)
    pack(w, s, w^L, l, w^R)
    return w

function getNodeL(h^τ, g, w, e)
    u ← sppfL(h^τ, g, w, e)
    (s, w′, _) ← h^τ
    A ← lhs(rule(s))
    l^L ← leftExtent(w′)
    l^R ← rightExtent(w)
    w ← sppf(A, l^L, l^R, e)
    ℒ ← ℒ ∪ {(w, u)}
    return w
```

Algorithm B.11 Function to construct an SPPF packed node.

```
function pack(w, s, w^L, l, w^R)
    if (s, l) ∉ nodes(w) then
        v ← (s, l)
        leftChild(v) ← w^L
        rightChild(v) ← w^R
        connect(w, v)
```

Algorithm B.12 Function to get a mapping from a linked list node to a set of sets of candidate terminals.

```
function getCandMap()
    φ ← an empty mapping
    for d ∈ I do
        (s, _, l, _, _, e) ← d
        a ← getCandidates(right(s), l, e)
        if l ∉ domain(φ) then
            φ(l) ← {}
        φ(l) ← φ(l) ∪ {a}
    return φ
```

Algorithm B.13 Function to set lookaheads for incomplete descriptors.

```
function setLookaheads(ψ)
    for d ∈ I do
        (s, g, l, _, w, e) ← d
        a ← getCandidates(right(s), l, e)
        if l ∈ domain(ψ) then
            for o ∈ ψ(l) do
                if o ∈ a then
                    add(s, g, l, o, w, e)
```

Algorithm B.14 Function to get a set of candidate terminals.

```
function getCandidates(b, l, e)
    if b ∈ N^G then
        a ← FIRST(b)
    else if b ∈ N^P then
        if b ∈ domain(Σ) then
            σ ← Σ(b)
            a ← σ(l, e)
        else
            a ← FIRST(b)
    else
        a ← {b}
    return a
```

Algorithm B.15 Function to compute scores for a model to predict terminals.

```
function computeScores(θ, l, x, c)
    c ← a vector ∈ ℝ^|T|
    for a ∈ T do
        c_ID(a) ← log p_θ(a | l, x) + c
    return c
```

Algorithm B.16 Function to get a unified mask vector.

```
function getUnifiedMask(â, M)
    m ← a vector ∈ ℝ^|T|
    for i ← 1 to |T| do
        m_i ← −∞
    for a ∈ â do
        if a ∈ domain(M) then
            m ← max(m, M(a))
        else
            for a ∈ a do
                m_ID(a) ← 0
    return m
```

Algorithm B.17 Function to create a mask vector.

```
function createMask(a)
    m ← a vector ∈ ℝ^|T|
    for i ← 1 to |T| do
        m_i ← −∞
    for a ∈ a do
        m_ID(a) ← 0
    return m
```

Table D.1: Examples of semantic parsing. We represented a linked list node l as the terminal sequence l stores

x :	There is a tower with exactly two blocks having a black block at the top
l :	`<s> exist filter all-boxes lambda x and = 2 count x exist filter x lambda y and is-top y is-black y</s>`
$\rho(\delta(\omega(l)))$:	`(exist (filter all-boxes (lambda (x) (and (= 2 (count x)) (exist (filter x (lambda (y) (and (top? y) (black? y))))))))))`
x :	there is a black square touching the edge
l :	`<s> exist filter all-items lambda x and is-black x and is-square x is-touching-wall x side.any</s>`
$\rho(\delta(\omega(l)))$:	`(exist (filter all-items (lambda (x) (and (black? x) (and (square? x) (touching-wall? x side.any))))))`
x :	There is 1 tower with a blue block at the top
l :	`<s> = 1 count filter all-boxes lambda x exist filter x lambda y and is-top y is-blue y</s>`
$\rho(\delta(\omega(l)))$:	`(= 1 (count (filter all-boxes (lambda (x) (exist (filter x (lambda (y) (and (top? y) (blue? y))))))))))`