Pre^3: Enabling Deterministic Pushdown Automata for Faster Structured LLM Generation

Junyi Chen[1*], **Shihao Bai**[2,3], **Zaijun Wang**[3], **Siyu Wu**[2], **Chuheng Du**[1], **Hailong Yang**[2], **Ruihao Gong**[2,3†], **Shengzhong Liu**[1†], **Fan Wu**[1], **Guihai Chen**[1]
[1]Shanghai Jiao Tong University [2]Beihang University [3]Sensetime Research
{junyi.chen, dch7723, shengzhong}@sjtu.edu.cn,
{wusiyu, hailong.yang, gongruihao}@buaa.edu.cn,
{wangzaijun, baishihao}@sensetime.com, {fwu,gchen}@cs.sjtu.edu.cn

Abstract

Extensive LLM applications demand efficient structured generations, particularly for LR(1) grammars, to produce outputs in specified formats (*e.g.*, JSON). Existing methods primarily parse LR(1) grammars into a pushdown automaton (PDA), leading to runtime execution overhead for context-dependent token processing, especially inefficient under large inference batches. To address these issues, we propose Pre^3 that exploits deterministic pushdown automata (DPDA) to optimize the constrained LLM decoding efficiency. First, by **pre**computing **pre**fix-conditioned edges during the **pre**processing, Pre^3 enables ahead-of-time edge analysis and thus makes parallel transition processing possible. Second, by leveraging the prefix-conditioned edges, Pre^3 introduces a novel approach that transforms LR(1) transition graphs into DPDA, eliminating the need for runtime path exploration and achieving edge transitions with minimal overhead. Pre^3 can be seamlessly integrated into standard LLM inference frameworks, reducing time per output token (TPOT) by up to 40% and increasing throughput by up to 36% in our experiments. Our code is available at https://github.com/ModelTC/lightllm.

1 Introduction

The recent remarkable development of Large Language Models (LLM) has ushered in new opportunities for a wide array of intelligent applications such as automated external tool invocations through function calls (Cai et al., 2023; Li et al., 2024a; Zhuo et al., 2024), chain of thoughts (Wei et al., 2022; Wang et al., 2022; OpenAI, 2024; Guo et al., 2025), embodied AI (Duan et al., 2022; Brohan et al., 2023; Yang et al., 2024b) et al. These applications created substantial demand for LLM systems to perform structured generation and produce outputs adhering to specific formats, such as JSON or other structures. Notably, major LLM API providers such as OpenAI and Alibaba Cloud now support JSON mode output to ensure deterministic schema compliance. Downstream applications can accordingly utilize these structured outputs to engage in downstream system interactions (Cho et al., 2023).

Constrained decoding (Hu et al., 2019; Scholak et al., 2021) is a widely used method in structured generation tasks (Willard and Louf, 2023b; Dong et al., 2023; Rückstieß et al., 2024) that excludes invalid tokens at each step by applying a *probability mask* to zero out their sample possibility. Flexible mechanisms like LR(1) grammars (Francis, 1961; Knuth, 1965a) are often employed to handle diverse and complex structural constraints, as they allow recursive rule definitions that surpass the limitations of regular expressions. However, this flexibility comes at the cost of degraded efficiency: Each decoding step requires parsing the grammar for all candidate tokens in a potentially large vocabulary. Additionally, tokens generated by LLM may consist of multiple characters that span across grammar rule boundaries, further complicating the generation process and demanding dedicated execution stack management. Both of them lead to significant computational overhead. These challenges raise the need to optimize constrained decoding efficiency without affecting LLM generation fidelity, making it more applicable in real-world applications.

Current state-of-the-art (SOTA) methods for constrained decoding acceleration, such as XGrammar (Dong et al., 2024), primarily focus on parsing LR(1) grammars into a pushdown automaton (PDA) (Nederhof and Satta, 1996). A PDA consists of multiple finite state automata (FSA), each representing a grammar rule, with the stack handling recursive rule expansions. These methods achieve substantial speedups by precomputing masks while

[*]Work done during the internship at Sensetime Research.
[†]Corresponding authors.

managing transitions through pushdown automata. However, they overlook the inherent properties of LR(1) grammars, which can be equivalently transformed into a deterministic pushdown automaton (DPDA) (Valiant, 1973, 1975).

The primary issue with traditional PDA-based approaches (Koo et al., 2024; Park et al., 2025a; Dong et al., 2022; Willard and Louf, 2023a; Li et al., 2024b) stems from the non-deterministic nature of the PDA's edges. Although these methods precompute masks based on the PDA structure, this design introduces two critical limitations. First, the non-deterministic edges depend on runtime contextual information to resolve transitions, resulting in incomplete precomputed masks for *context-dependent tokens*. The computation of context-dependent tokens necessitates backtracking, speculative operations, and the maintenance of a *persistent stack* (merges all past stacks into a tree, with each stack as a root-to-node path) during runtime. As batch sizes increase, the overhead from these runtime computations grows significantly, severely degrading decoding efficiency. Second, previous methods cannot effectively optimize non-deterministic transitions during preprocessing because they will dynamically change during runtime. This limitation hinders their ability to fully exploit the potential of the parsing method, leading to suboptimal performance.

To address these challenges, we propose Pre^3, a constrained LLM decoding approach based on a deterministic pushdown automaton (DPDA). Unlike traditional methods, we design an algorithm to directly build a DPDA from the LR(1) grammar. Leveraging the deterministic nature of the DPDA's edges, our approach resolves the aforementioned limitations. First, the determined transitions in the DPDA eliminate the context-dependent tokens, further entirely eliminating the need for backtracking, speculative exploration, and the maintenance of a persistent stack. This fundamentally reduces the runtime computational overhead associated with transitions. Second, since all transition edges in the DPDA are available during preprocessing, we can perform comprehensive optimizations on the automaton in advance. Additionally, for the stack-matched transition mechanism of the DPDA, we design a parallel verification method for transitions, which accelerates inference. Together, these innovations result in a more efficient and scalable constrained decoding framework.

In summary, the paper's main contributions are:

Table 1: Per token latency comparison (in milliseconds) across different batch sizes.

Batch Size	**16**	**32**	**64**	**128**	**256**	**512**
Baseline	11.38	21.87	25.74	30.08	56.29	92.23
XGrammar	15.19	43.69	52.07	65.21	90.98	147.64

- We first propose an algorithm to transform LR(1) state transition graphs into DPDA, eliminating runtime exploration and enabling edge transitions with minimal overhead.
- We enable additional optimizations for edges and support parallel transition processing by precomputing prefix-conditioned edges.
- We integrate Pre^3 into mainstream LLM inference systems and achieve up to 40% improvement in time per output token (TPOT) and increase throughput by up to 36% with high scalability into large batch sizes.

2 Preliminaries and Background

2.1 LLM Constrained Decoding

Constrained decoding (Hu et al., 2019) enforces strict grammatical adherence by dynamically pruning invalid tokens during generation. While effective for structural compliance, existing methods face two key challenges: (1) handling diverse grammars, large vocabularies, and complex token-to-text mappings, and (2) computational inefficiency at scale, especially under large batch processing where dynamic validation creates sequential bottlenecks.

Our empirical analysis reveals this critical limitation. When evaluating XGrammar on Meta-Llama-3-8B (2×H800 GPUs), constrained decoding exhibits up to 37.5% higher latency (147.64 ms vs. 92.23 ms) at batch size 512 compared to unconstrained decoding (Table 1). The performance gap grows with batch size due to non-parallelizable validation steps, which is a fundamental constraint in current approaches. These findings motivate the need for a new constrained decoding approach that maintains grammatical correctness while achieving better computational efficiency at scale.

2.2 LR(1) Grammar and State Transition Graphs

In constrained decoding scenarios, most grammars can be classified as LR(1) grammars, which are fundamental to bottom-up parsing and align naturally with the token-by-token generation process of large language models (LLMs). LR(1) grammars are a powerful subset of context-free grammars capable of describing the syntax of most programming languages. They are characterized by their ability to handle deterministic parsing with a single lookahead symbol, making them highly expressive and widely applicable. Nearly all context-free grammars can be converted into LR(1) form, which ensures their versatility in modeling structured languages. This property, combined with their alignment with bottom-up parsing methods, makes LR(1) grammars a cornerstone in constrained decoding and syntactic analysis tasks.

LR(1) items are tuples of the form $[A \to \alpha \cdot B\beta, a]$, where $A \to \alpha \cdot B\beta$ represents the parsing progress of a production rule, and a is a lookahead symbol used to determine when a reduction should occur. The `CLOSURE` operation constructs LR(1) item sets by adding items for non-terminals and their productions, ensuring all possible derivations are considered. The `GOTO` function generates the LR(1) state transition graph by moving the dot in items past a grammar symbol X and computing the closure of the resulting items, thereby connecting states to form the LR(1) automata. This process continues until no new states are generated, creating a complete parsing structure for the grammar.

2.3 Deterministic Pushdown Automata (DPDA)

Pushdown automata (PDA) are a class of abstract machines that extend finite automata with an unbounded stack memory, enabling them to recognize context-free languages (CFLs) (Hopcroft et al., 2001). A PDA is defined as a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where Q is a finite set of states, Σ is the input alphabet, Γ is the stack alphabet, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the transition function, q_0 is the initial state, Z_0 is the initial stack symbol, and $F \subseteq Q$ is the set of accepting states. The non-deterministic transition function δ allows PDAs to handle ambiguous structures inherent to context-free grammars, *e.g.*, nested parentheses or recursive syntactic patterns.

A deterministic pushdown automaton (DPDA) is a restricted variant where, for every state $q \in Q$, input symbol $a \in \Sigma$, and stack symbol $Z \in \Gamma$, the transition function $\delta(q, a, Z)$ yields at most one possible move, and ϵ-transitions (stack operations without consuming input) are permitted only if no input-consuming transition is available (Sipser, 1996). This determinism ensures unique computation paths, making DPDAs equivalent to the class of deterministic context-free languages (DCFLs), which are unambiguous and efficiently parsable. As mentioned earlier, the vast majority of grammars in the constrained decoding scenario can be represented by LR(1), which is a true subset of DCFL and can be recognized by DPDA (ASU86 et al., 1986; Sipser, 1996). Compared to PDA, DPDA avoided backtracking and non-deterministic search overhead, which can significantly improve the efficiency of constrained decoding. See Appendix A for additional background on formal language theory.

3 Pre^3 Design

Our proposed method, Pre^3, is a DPDA-based constrained decoding solution that leverages a novel approach for constructing a DPDA from a given LR(1) grammar. The method operates by first transforming the LR(1) grammar into an LR(1) state transition graph, which is then converted into a DPDA using the techniques introduced in this section. This DPDA can be directly utilized for constrained decoding, enabling efficient and effective decoding. The method requires only minimal processing time, averaging 3-5 seconds for complex JSON grammars and under 0.1 seconds for simpler grammars (*e.g.*, arithmetic expressions). Notably, this is a one-time cost as the results are cacheable and reusable. The complete workflow of our method is illustrated in Figure 1.

In Section 3.1, we introduce the Prefix-conditioned Edge, a novel mechanism ensuring uniqueness by matching both prefix information and input symbols, unlike traditional PDA transitions. In Section 3.2, we design an algorithm to compute all LR(1) state transitions, incorporating Prefix-conditioned Edge and addressing cyclic structures, successfully constructing a DPDA. In Section 3.3, we optimize the DPDA's structure and performance through preprocessing, leveraging its pre-determined edges.

3.1 Prefix-conditioned Edges

Constrained decoding with LLMs faces challenges due to non-deterministic transitions in PDA, where the same input symbol can trigger multiple transitions based on prior symbol sequences. This non-determinism complicates computation by requiring speculative exploration, backtracking, and a persistent stack to store historical context, increasing overhead. To resolve these issues, eliminating non-determinism in transitions is crucial for enabling

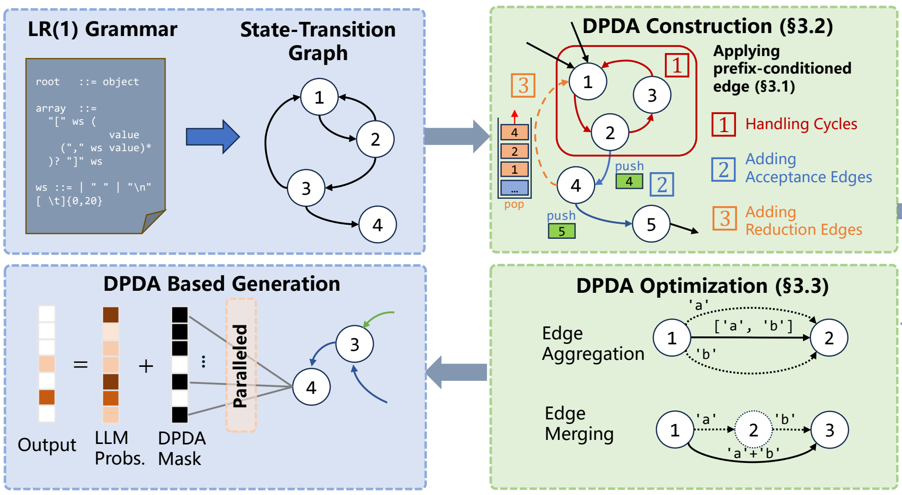

Figure 1: Overview of Pre[3]: The figure depicts the workflow from LR(1) grammar to DPDA-based generation, encompassing DPDA construction and optimization steps.

preprocessing optimizations and efficient runtime computation.

A fundamental property of LR(1) grammars is that **the current stack configuration and a single lookahead symbol are sufficient to uniquely determine the next action**. This property provides a theoretical foundation for introducing determinism into the automaton's transition edges. Building on this insight, we propose the Prefix-conditioned Edge, as illustrated in Figure 2.

By simultaneously considering the input symbol and the prefix of accepted symbols (represented by the stack's state), we uniquely determine the target state for each transition. To achieve this, our method enhances each edge with three key components:

- **Accepted Symbol**: The input symbol that triggers the transition.
- **Stack Matching Condition**: The specific prefix of the stack required for the transition to be valid.
- **Stack Operations**: Actions such as push to update the stack during the transition, which is both required by PDAs and DPDAs.

Notably, although the additional stack-matching conditions introduced to the edges increase complexity, we address this challenge by implementing a parallel algorithm capable of simultaneously verifying multiple stack-matching conditions, effectively resolving this issue.

3.2 Cycle-aware Deterministic Pushdown Automata Construction

To avoid the additional exploration overhead at runtime, we aim to construct a DPDA based on LR(1) grammars. However, building a DPDA is non-trivial and requires a systematic approach. In this section, we introduce our algorithm for constructing a DPDA from an LR(1) state transition graph step by step, leveraging the prefix-conditioned edge to ensure determinism.

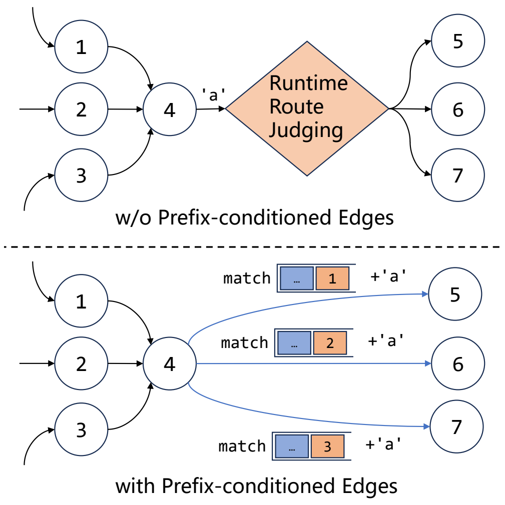

Figure 2: This diagram illustrates prefix-conditioned edges: above shows the case before calculation, where ‘a’ is a context-dependent token requiring runtime context for transition; below shows the precomputed case, where each edge includes a stack-matching condition, uniquely determining the transition path via the condition and transition symbol.

3.2.1 DPDA Structure

We begin our algorithm with the state transition graph generated from the LR(1) grammar, where the nodes represent the LR(1) item set family and the edges indicate the acceptance of a symbol when traversing from one node to another. Building on this foundation, we construct the DPDA by retaining the node definitions from the LR(1) transition graph but redefining the edges into two distinct

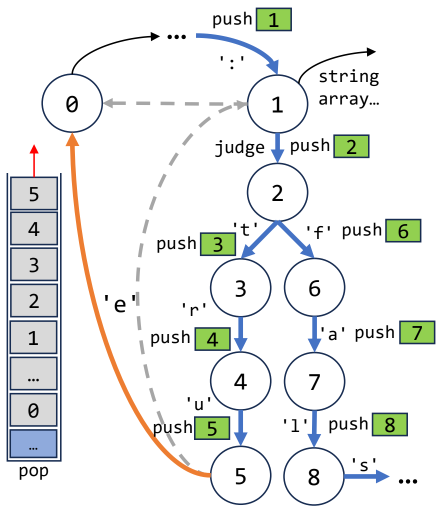

Figure 3: Two edge types for DPDA computation: blue edges are acceptance edges (existing in the original LR(1) graph, handling stack operations for acceptance); orange edges are reduction edges (added to the DPDA, matching and popping stack operations for reductions); gray edges depict LR(1) reduction paths, demonstrating fewer nodes needed for reduction after state machine construction.

types: *acceptance edges* and *reduction edges*, as shown in Figure 3.

- **Acceptance Edges** are the simplest type of transition in our DPDA. These edges are directly derived from the original state transition graph of the LR(1) grammar. In the context of LR(1) parsing, an acceptance edge corresponds to a shift operation, where the automaton consumes an input symbol from the input stream and pushes it onto the stack while transitioning to a new state. This operation reflects the fundamental step of recognizing and accepting a terminal symbol in the input, advancing the parsing process.

- **Reduction Edges** model reduction operations in LR(1) parsing. In traditional LR(1) parsing, reductions involve replacing a sequence of terminal symbols with a non-terminal symbol according to the grammar rules. However, nested grammar rules often require multiple reduction steps, leading to inefficiencies. Reduction edges address this by directly encoding reduction operations as single-step transitions during the pre-processing phase. These edges connect reduction targets, enabling the automaton to handle nested reductions efficiently.

3.2.2 Acceptance Edges and Reduction Edges Integration

The state transition graph alone cannot function as a DPDA because it only supports shift operations (*i.e.*, symbol acceptance) and lacks reduction operations, while some edges also suffer from nondeterminism. To address these issues, we not only compute all possible transition edges, including both shift and reduction edges, to complete the missing reduction paths, but also leverage prefix-conditioned edges to incorporate stack conditions into each transition, resolving nondeterminism and enabling the transformation of the nondeterministic state transition graph into a DPDA.

Adding Acceptance Edges: Acceptance edges do not need to consider determinism because the construction of the LR(1) state transition graph ensures that no node will have two identical transitions. As a result, when an acceptance edge is encountered, the target node's state information is simply pushed onto the runtime stack. The algorithmic flow of this operation is described in Lines 6–8 of Algorithm 1.

Adding Reduction Edges: Based on the definition of reduction edges, we can employ a two-step method to add all necessary reduction edges to the automaton, which is described in Lines 9–18 of Algorithm 1.

First, we identify ϵ-reduction transitions, representing unconditional reductions, and add them to the automaton to handle mandatory reductions. These transitions backtrack along their path, popping states until reaching the reduction endpoint. However, their lack of accepted symbols introduces ambiguity, violating the DPDA's determinism. To ensure completeness, this process is applied recursively, generating all necessary reduction edges by traversing the state transition graph.

Second, we resolve indeterminism by merging ϵ-reduction edges with compatible acceptance edges, ensuring aligned stack operations and reduction targets, and assigning appropriate accept tokens to satisfy the Prefix-condition.

3.2.3 Solving Issues with Automaton Cycles

LR(1) grammars are highly expressive and can handle complex language constructs, including the acceptance of cyclic symbol sequences. However, cycles introduce significant challenges when constructing a DPDA.

During the precomputation of reduction edges, cycles create a critical issue: repeatedly traversing a cycle generates an infinite number of potential reduction paths. This makes it computationally infeasible to add all necessary reduction edges. Figure 4 visually illustrates how cycles in the automaton can

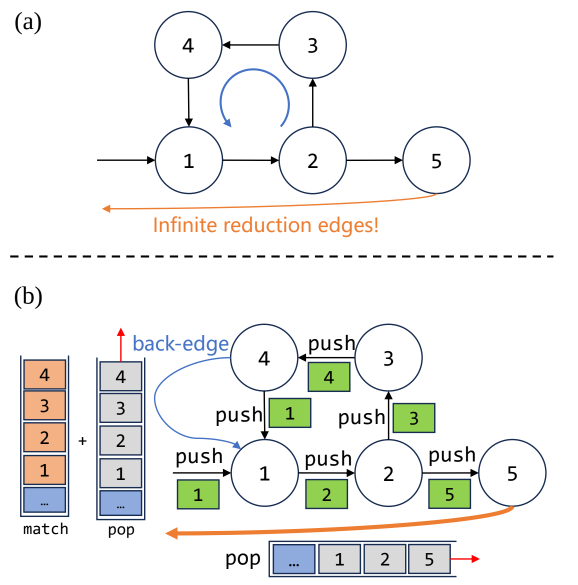

Figure 4: (a) Pushdown automaton with an infinite cycle between State 1, 2, 3, 4, leading to an infinite number of possible paths and indeterminable transition paths when adding reduction edges at State 5; (b) How our method handles the cycle issue: The back-edge from State 4 to State 1 is modified to check for complete cycle traversal information (*e.g.*, [1, 2, 3, 4]) in the stack. If detected, it pops the redundant state (*e.g.*, [1, 2, 3, 4]), ensuring reduction edges at State 5 only need to account for traversals without cycles.

lead to infinite reduction paths.

Through further observation, we note that during the reduction process, specifying an entry node and an exit node uniquely determines the path along which the reduction occurs. This property allows us to disregard the number of cycle traversals, as even a single traversal of the cycle does not need to be explicitly recorded.

We propose a solution that simplifies the reduction process as follows: Suppose we have a detected cycle with the reduction problem $C = (s_1, s_2, s_3, \dots, s_n, s_1)$. We define the *back-edge* as $s_n \to s_1$. While handling the cycle, we modify this back-edge by introducing an additional stack operation: a pop operation for the sequence $(s_1, s_2, \dots, s_n)$. This modification enables efficient handling of cyclic traversals.

Furthermore, by checking whether all vertices traversed in a single cycle are fully present in the execution stack, we ensure that the stack retains only the necessary information from outside the cycle traversal. Specifically, if a complete traversal of the cycle is detected, the stack information corresponding to the current traversal is popped

Algorithm 1: Construct DPDA from LR(1) Transition Graph

```
Input: LR(1) State Transition Graph $G = (S, E)$
Output: Deterministic Pushdown Automata (DPDA)
/* Step 1: Cycle Handling                                  */
1  $C$ ←Detect cycles with reduction problem in $G$
2  foreach detected cycle $C = (s_1, s_2, ..., s_n, s_1)$ do
3      if $C$ corresponds to recursive reduction of non-terminal $A$ then
4          Define the back-edge: $s_n \xrightarrow{\text{back}} s_1$
5          Modify the back-edge to check for complete cycle traversal in the stack: match and pop $(s_1, s_2, ..., s_n)$, push($s_1$)
/* Step 2: Acceptance Edge Generation                      */
6  foreach state $s_i \in S$ do
7      foreach valid transition $s_i \xrightarrow{X} s_j$ in $E$ do
8          Add stack operation: push($s_j$)
/* Step 3: Reduction Edge Generation                       */
9  Function GenerateReductionEdges(state $s_i$):
10     foreach reduction sequence $s_i \xrightarrow{\text{reduce } A} s_j \xrightarrow{\text{reduce } B} s_k$ do
11         Merge into a direct transition: $s_i \xrightarrow{\text{reduce } A \to B} s_k$
12         Validate stack compatibility
13         GenerateReductionEdges($s_k$)
14     foreach $\epsilon$-reduction edge from $s_i$ do
15         Merge the $\epsilon$-reduction edge with appropriate acceptance edges that share the same stack operations
16         Assign suitable accept tokens to ensure the Prefix-condition is matched
17         GenerateReductionEdges(target state of the merged edge)
18 GenerateReductionEdges(initial state $s_0$)
```

immediately. This guarantees that the stack never accumulates redundant context from repeated cycle traversals.

This approach, described in Algorithm 1, lines 1–5, guarantees that the system reverts to an equivalent state after each complete traversal, avoiding infinite reduction edges. As a result, the automaton can handle cycles efficiently without compromising determinism or computational feasibility.

3.3 Edge Optimization with Prefix-condition

Building on the DPDA constructed in Section 3.2, we can further perform various optimizations. Since all transition edges in the DPDA are deterministic and can be uniquely resolved by matching both the stack state and input symbols, we are able to analyze the automaton's structure during the preprocessing phase. In contrast, traditional methods based on non-deterministic pushdown automata (PDA) cannot achieve such optimizations during preprocessing due to the ambiguity of transition edges, where the same input symbol may lead to multiple possible transition targets. For example,

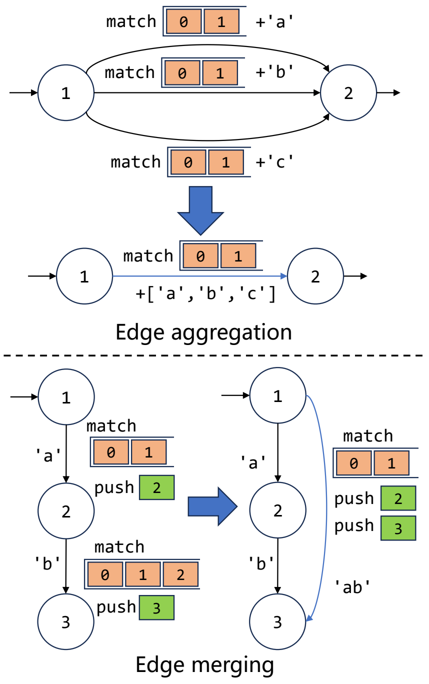

Figure 5: Two different types of edge optimization.

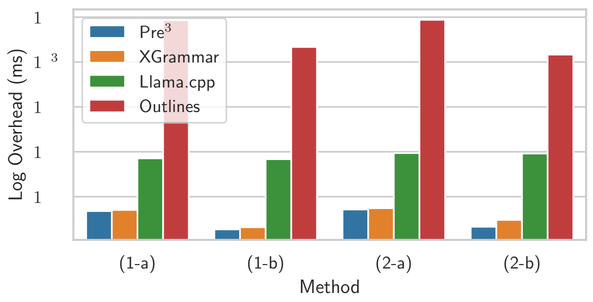

Figure 6: Per-step decoding overhead across different grammars and models. Outlines incurs an overhead of up to several seconds per step. Experiments contain (1) Chain-of-Thought grammar, (2) JSON grammar. Models contain (a) Meta-Llama-3-8B on 1×H800, (b) Meta-Llama-2-70B on 4×H800.

we can aggregate and merge transition edges as shown in Figure 5.

- **Edge Aggregation:** Edges with the same stack prefix condition and stack operations but different accepted symbols can be combined. For example, in grammars describing numbers, edges for digits 0-9 can be merged into a single edge accepting all digits to simplify the automaton.

- **Edge Merging:** If two edges share the matched stack prefix condition and operations, we can connect them directly, and add a new edge. This is important for LLM constrained decoding scenarios, as it allows to "jump" to the desired state in fewer steps, reducing the scale of the DPDA.

These operations are enabled by precomputed prefix-conditioned edges for all stack conditions. Without prefix-conditioned edges, transitions that depend on dynamic stack inspection cannot be analyzed in advance.

4 Evaluation

4.1 Experimental Setup

Implementation: We implemented our approach in 2,000 lines of Python code and about 1,000 lines of C++ code, and we seamlessly integrated with LightLLM (Gong et al., 2025), a popular LLM inference framework.

Hardware Setup: All the experiments are tested on a server with Intel(R) Xeon(R) Gold 6448Y CPU and 8 NVIDIA H800 GPUs. Depending on the scale of the experiment, we use different numbers of GPUs.

Baselines: We choose the following representative works on grammar constraint decoding.

- **XGrammar:** An open-source library for structured generation in large-language models. It significantly enhances performance in tasks like JSON grammar generation with reduced latency and storage.
- **Outlines:** A text generation library, it offers a Python tool for grammar-guided generation, offering a fast generation method. We use vLLM integrated with Outlines for evaluation.
- **Llama.cpp**: A C/C++-based LLM inference tool, and also includes support for grammar constraint decoding.

Datasets: In our experiments, we utilized the JSON-mode-eval (NousResearch, 2024) dataset from NousResearch as prompts. As there is a scarcity of datasets for structured output, we collected some private data additionally and incorporated it into the test dataset.

4.2 Per-step Decoding Efficiency

To evaluate the improvement of our system, we first examine the per-step decoding overhead, which is defined by subtracting the original decoding time from the grammar-based decoding time. We design four experiment setups, including two models, Meta-Llama-3-8B and Meta-Llama-2-70B, and two grammars with English characters, JSON and chain-of-thought. For comparison, we benchmark our method against several state-of-the-art and popular structure generation engines, including XGrammar, Outlines, and llama.cpp, to demonstrate the efficiency of our system at a per-step scale.

The results are shown in Figure 6 and Table 3. Pre^3 demonstrates a superiority over Outlines and llama.cpp, and Pre^3 remains a consistent advantage over XGrammar. The results indicate that Pre^3 introduces less overhead than previous SOTA systems.

4.3 Large Batch Inference Efficiency

In real-world serving scenarios, inference often handles large batches of requests simultaneously, making large-batch efficiency crucial for deploying language models at scale. We evaluate performance in such settings, where efficiency gains significantly impact system performance.

We benchmark Pre^3 against the state-of-the-art XGrammar, using the JSON grammar for its complexity and challenging recursive structures (*e.g.*, lists and dictionaries). This tests the robustness and scalability of our method under demanding conditions.

Our experiments are conducted on multiple models of varying sizes and architectures. Specifically, we conducted experiments on Llama3-8B and Deepseek-V2 (15.7B) on a $2\times$H800 setup, and Llama2-70B on a $4\times$H800 setup. The maximum batch size goes to 1024, large enough to test the scalability of our method. In this experiment, we also measured the average time taken for each step, but the requests are batched in number to test the system's ability to process large batches.

The result is shown in Table 2. The results show that Pre^3 consistently outperforms XGrammar in all scenarios with latency reduction by up to 30%. The advantage is more significant at larger batch sizes, demonstrating the scalability of Pre^3.

4.4 Realworld Deployment

To evaluate the throughput in real-world service environments, we compare the performance of XGrammar and Pre^3, under varying system concurrency levels. We conducted simulation experiments on Meta-Llama-3-8B ($2\times$H800) and Meta-Llama-2-70B ($4\times$H800), measuring the throughput in burst scenarios at different levels of concurrency.

The results are shown in Figure 7. Both Pre^3 and XGrammar have lower throughput than the Original system due to the added overhead introduced by constraint decoding, while Pre^3 demonstrated a significant improvement over XGrammar, achieving up to 20% higher throughput at higher concurrency levels, showing that Pre^3 provides higher throughput in end-to-end deployment.

5 Related Work

LLM Constrained Decoding. Several approaches have been proposed for constrained decoding in language models, yet most exhibit limitations when applied to large-batch inference. Implementations such as llama.cpp (Gerganov, 2023) rely on inefficient runtime token verification, introducing significant computational overhead. Subsequent methods like Outlines (Willard and Louf, 2023b) and SynCode (Ugare et al., 2024) improve upon grammar-guided generation but still face suboptimal decoding efficiency. The current state-of-the-art method, XGrammar (Dong et al., 2024), achieves impressive speed for small batch sizes; however, its performance degrades as batch sizes increase due to growing computational overhead. Similarly, GreatGrammar (Park et al., 2025b) demonstrates strong efficiency in handling complex grammars but is only evaluated with batch size equals to 1, leaving its scalability to larger batches an open question.

LR(1) Grammar Parser. The theoretical foundations of LR(1) parsing and its equivalence to deterministic pushdown automata (DPDA) have been well-established in formal language theory and compiler design (DeRemer, 1969; Lehmann, 1971; Korenjak, 1969). Traditional LR(1) parsers, such as those described by Knuth (Knuth, 1965b), use state-merging techniques to construct minimal parsing tables, enabling deterministic recognition of context-free languages. Recent work, including IELR(1) (Denny and Malloy, 2008) and PSLR(1) (Denny, 2010), further optimized the parser by addressing state conflicts and improving efficiency in handling composite grammars. These methods ensure that LR(1) parsers can be systematically converted into DPDA implementations, where a deterministic state transition table guides stack operations. While prior work has focused on compiler parsing, applying LR(1)-to-DPDA techniques to constrained decoding in large language models (LLMs) poses unique challenges. To our knowledge, Pre^3 presents the first adaptation of LR(1) parsing techniques to the domain of LLM constrained decoding.

6 Conclusion

This work addressed the limitations of existing structured generation approaches by proposing a DPDA-based methodology (Pre^3), which integrates

Table 2: Decode batch inference time comparison between our method and XGrammar. The “-” marker stands for the batch size cannot be executed on the given hardware setup.

Evaluation Configuration	Method	16	32	64	128	256	512	1024
Llama-3-8B (Dubey et al., 2024)	XGrammar (ms)	15.19	43.69	52.07	65.21	90.98	147.64	272.77
	Pre^3 (ms)	11.77	31.12	35.88	45.32	64.42	104.46	201.16
2×H800	Reduction	**↓22.49%**	**↓28.78%**	**↓30.09%**	**↓30.50%**	**↓29.20%**	**↓29.24%**	**↓26.25%**
DeepSeek-V2-Lite-Chat (Liu et al., 2024)	XGrammar (ms)	51.76	59.45	77.74	104.06	121.46	-	-
	Pre^3 (ms)	49.91	53.71	54.41	61.63	75.47	-	-
15.7B 2×H800	Reduction	**↓3.57%**	**↓9.65%**	**↓30.01%**	**↓40.78%**	**↓37.86%**	-	-
Qwen2-14B (Yang et al., 2024a)	XGrammar (ms)	16.77	47.94	57.05	74.54	98.64	162.47	285.42
	Pre^3 (ms)	16.52	47.94	47.89	65.50	90.20	143.83	232.18
INT8 2×H800	Reduction	**↓1.52%**	**↓0.12%**	**↓2.37%**	**↓12.14%**	**↓8.55%**	**↓11.47%**	**↓18.65%**
Llama-2-70B (Touvron et al., 2023)	XGrammar (ms)	28.75	55.12	56.94	68.79	85.92	-	-
	Pre^3 (ms)	27.20	54.24	54.18	62.27	75.72	-	-
4×H800	Reduction	**↓5.39%**	**↓1.60%**	**↓4.85%**	**↓9.48%**	**↓11.87%**	-	-

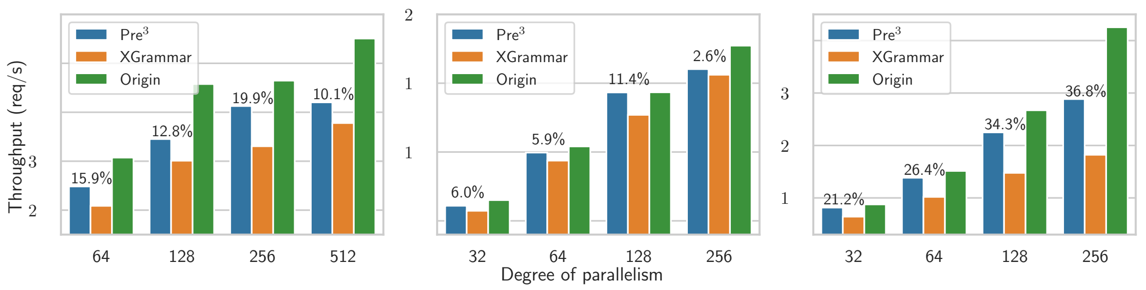

Figure 7: System throughput based on different models and concurrency levels. Left: Llama3-8B, Middle: Llama2-70B, Right: DeepSeek-V2-Lite-Chat.

Table 3: Per-step decode time comparison between our method and XGrammar.

	Llama-3-8B		Llama-2-70B	
Batchsize	Pre^3	XGrammar	Pre^3	XGrammar
1	**0.5172**	0.5531	**0.2163**	0.3030
4	**0.6537**	0.9327	**0.2407**	0.3310

Cycle-aware Deterministic Pushdown Automata Construction and Prefix-conditioned Edge Optimization. Pre^3 significantly outperforms existing SOTA baselines by up to 40% in throughput and exhibits greater advantages with large batch sizes.

Limitation

While our work demonstrates significant improvements in constrained LLM decoding efficiency, several limitations and potential areas for improvement remain. Firstly, our method is optimized for LR(1) grammars, which cover most structured generation needs, but faces challenges with more complex LR(k) grammars ($k > 1$). These require intricate state transitions and lookahead mechanisms, increasing DPDA construction and processing complexity. Future work should explore hybrid parsing or adaptive mechanisms to handle such cases efficiently. Second, our Python-based research prototype lacks production-level optimizations. Although suitable for experimentation, the implementation could benefit from hardware acceleration (*e.g.*, GPU parallelization for grammar processing) and system-level optimizations. A future C++/Rust implementation with fine-tuned memory management could significantly improve performance and scalability. Although preprocessing complexity scales with grammar size (due to increased edges and cycles), current processing times remain practical for real-world deployment. Addressing these limitations could unlock additional performance improvements and broaden the applicability of our approach.

Acknowledgements

This work was partially sponsored by the National Key R&D Program of China (No. 2022ZD0119100), in part by China NSF grant No. 62472278, 62025204, 62432007, 62441236, 62332014, 62332013, and 62322201, in part by Alibaba Group through Alibaba Innovation Research Program, and in part by Tencent Rhino Bird Key Research Project. This work was partially supported by SJTU Kunpeng & Ascend Center of Excellence. The opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies or the government.

References

AV ASU86, R Sethi Aho, and Ullman JD. 1986. Compilers: Principles, techniques, and tools.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. 2023. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2023. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*.

Sukmin Cho, Soyeong Jeong, Jeong yeon Seo, and Jong Park. 2023. Discrete prompt optimization via constrained generation for zero-shot re-ranker. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 960–971, Toronto, Canada. Association for Computational Linguistics.

Joel E Denny. 2010. *PSLR (1): Pseudo-scannerless minimal LR (1) for the deterministic parsing of composite languages*. Ph.D. thesis, Clemson University.

Joel E Denny and Brian A Malloy. 2008. Ielr (1) practical lr (1) parser tables for non-lr (1) grammars with conflict resolution. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 240–245.

Franklin Lewis DeRemer. 1969. *Practical translators for LR (k) languages*. Ph.D. thesis, Massachusetts Institute of Technology.

Yihong Dong, Xue Jiang, Yuchen Liu, Ge Li, and Zhi Jin. 2022. Codepad: Sequence-based code generation with pushdown automaton. *arXiv preprint arXiv:2211.00818*.

Yihong Dong, Ge Li, and Zhi Jin. 2023. Codep: grammatical seq2seq model for general-purpose code generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 188–198.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2024. Xgrammar: Flexible and efficient structured generation engine for large language models. *Preprint*, arXiv:2411.15100.

Jiafei Duan, Samson Yu, Hui Li Tan, Hongyuan Zhu, and Cheston Tan. 2022. A survey of embodied ai: From simulators to research tasks. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 6(2):230–244.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

John GF Francis. 1961. The qr transformation a unitary analogue to the lr transformation—part 1. *The Computer Journal*, 4(3):265–271.

Georgi Gerganov. 2023. llama.cpp. LLM inference in C/C++.

Ruihao Gong, Shihao Bai, Siyu Wu, Yunqian Fan, Zaijun Wang, Xiuhong Li, Hailong Yang, and Xianglong Liu. 2025. Past-future scheduler for llm serving under sla guarantees. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 798–813, New York, NY, USA. Association for Computing Machinery.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

J. Edward Hu, Huda Khayrallah, Ryan Culkin, Patrick Xia, Tongfei Chen, Matt Post, and Benjamin Van Durme. 2019. Improved lexically constrained decoding for translation and monolingual rewriting. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 839–850, Minneapolis, Minnesota. Association for Computational Linguistics.

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

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

Terry Koo, Frederick Liu, and Luheng He. 2024. Automata-based constraints for language model decoding. *arXiv preprint arXiv:2407.08103*.

AJ Korenjak. 1969. A practical method for constructing lr (k) processors. *Communications of the ACM*, 12(11):613–623.

Daniel Lehmann. 1971. Lr (k) grammars and deterministic languages. *Israel Journal of Mathematics*, 10:526–530.

Zekun Li, Zhiyu Zoey Chen, Mike Ross, Patrick Huber, Seungwhan Moon, Zhaojiang Lin, Xin Luna Dong, Adithya Sagar, Xifeng Yan, and Paul A Crook. 2024a. Large language models as zero-shot dialogue state tracker through function calling. *arXiv preprint arXiv:2402.10466*.

Zelong Li, Wenyue Hua, Hao Wang, He Zhu, and Yongfeng Zhang. 2024b. Formal-llm: Integrating formal language and natural language for controllable llm-based agents. *arXiv preprint arXiv:2402.00798*.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Mark-Jan Nederhof and Giorgio Satta. 1996. Efficient tabular lr parsing. *arXiv preprint cmp-lg/9605018*.

NousResearch. 2024. Nousresearch/json-mode-eval.

OpenAI. 2024. Learning to reason with llms.

Kanghee Park, Timothy Zhou, and Loris D'Antoni. 2025a. Flexible and efficient grammar-constrained decoding. *arXiv preprint arXiv:2502.05111*.

Kanghee Park, Timothy Zhou, and Loris D'Antoni. 2025b. Flexible and efficient grammar-constrained decoding. *Preprint*, arXiv:2502.05111.

Thomas Rückstieß, Alana Huang, and Robin Vujanic. 2024. Origami: A generative transformer architecture for predictions from semi-structured data. *arXiv preprint arXiv:2412.17348*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*.

Michael Sipser. 1996. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Syncode: Llm generation with grammar augmentation. *Preprint*, arXiv:2403.01632.

Leslie Valiant. 1973. *Decision procedures for families of deterministic pushdown automata*. Ph.D. thesis, University of Warwick.

Leslie G Valiant. 1975. Regularity and related problems for deterministic pushdown automata. *Journal of the ACM (JACM)*, 22(1):1–10.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Brandon T Willard and Rémi Louf. 2023a. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*.

Brandon T. Willard and Rémi Louf. 2023b. Efficient guided generation for large language models. *Preprint*, arXiv:2307.09702.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Yijun Yang, Tianyi Zhou, Kanxue Li, Dapeng Tao, Lusong Li, Li Shen, Xiaodong He, Jing Jiang, and Yuhui Shi. 2024b. Embodied multi-modal agent trained by an llm from a parallel textworld. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 26275–26285.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Supplementary Materials on Formal Language Theory

To facilitate understanding of our algorithm, we introduce some basics of formal language theory in the supplementary materials.

A.1 Formal Definition of LR(1) Grammars

LR(1) grammars constitute a fundamental class of context-free grammars that can be parsed deterministically with one-symbol lookahead. Formally, a grammar $G = (V, \Sigma, P, S)$ is said to be LR(1) if the following conditions are satisfied:

- **Uniqueness of Valid Items:** For any viable prefix γ, there exists at most one LR(1) item of the form $[A \to \alpha \cdot \beta, a]$, where $a \in \Sigma \cup \{\$\}$, that is valid at that parsing configuration.

- **Reduction Consistency:** If two items $[A \to \alpha\cdot, a]$ and $[B \to \beta\cdot, b]$ are simultaneously valid for the same viable prefix γ, then at least one of the following must hold: $A \neq B$, $\alpha \neq \beta$, or $a \neq b$.

These constraints ensure that shift/reduce and reduce/reduce conflicts are resolved uniquely using the lookahead token. All LR(0) grammars are properly contained within the LR(1) class, and the parsing process maintains linear time complexity with respect to the input length.

A.2 Detailed Construction Procedure of the Canonical LR(1) Automaton

The construction of the canonical LR(1) automaton is a central step in generating a deterministic parser for a given context-free grammar. The automaton is a finite state machine in which each state represents a set of LR(1) items, and transitions correspond to the recognition of grammar symbols.

Let $G = (V, \Sigma, P, S)$ be the original grammar. We begin by augmenting it with a new start symbol $S' \notin V$, and add the production $S' \to S$. The augmented grammar is denoted by $G' = (V \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S')$.

LR(1) Item An LR(1) item is a pair $[A \to \alpha \cdot \beta, a]$, where:

- $A \to \alpha\beta$ is a production rule in P,
- The dot $(\cdot)$ indicates the current parsing position within the right-hand side,
- $a \in \Sigma \cup \{\$\}$ is a lookahead symbol, representing the terminal expected to follow the derivation of A.

Item Set (State) A state in the automaton is a set of LR(1) items, closed under the `CLOSURE` operation. Each state encapsulates a snapshot of all valid parser configurations at a certain point in the input.

Closure Operation The `CLOSURE` function expands a set of items I by recursively adding all items that could be expected next due to nonterminal symbols appearing after the dot. Formally:

$$\texttt{CLOSURE}(I) = \text{smallest superset of } I$$

such that:

$$\forall [A \to \alpha \cdot B\beta, a] \in I,\ \forall B \to \gamma \in P,\ \forall b \in \text{FIRST}(\beta a) :\ [B \to \cdot\gamma, b] \in \texttt{CLOSURE}(I)$$

The key detail here is that the lookahead symbol a propagates through the FIRST(βa) computation, ensuring context-sensitivity.

Goto Operation Given a state I and a grammar symbol $X \in V \cup \Sigma$, the `GOTO` function computes the next state by advancing the dot over X in all applicable items:

$$\texttt{GOTO}(I, X) = \texttt{CLOSURE}\left(\{[A \to \alpha X \cdot \beta, a] \mid [A \to \alpha \cdot X\beta, a] \in I\}\right)$$

This represents the parser consuming symbol X and transitioning to a new parser configuration.

Construction Algorithm Let C denotes the set of item sets (states). The construction algorithm proceeds as follows:

1. Initialize:
$$I_0 = \texttt{CLOSURE}(\{[S' \to \cdot S, \$]\}), \quad C := \{I_0\}$$
2. Iteratively expand:
For each $I \in C$, and each grammar symbol $X \in V \cup \Sigma$:
$$J := \texttt{GOTO}(I, X)$$
If $J \neq \emptyset$ and $J \notin C$, then add J to C and record a transition:
$$\delta(I, X) = J$$
3. Repeat until no new item sets are added to C.

Parsing with the Canonical LR(1) Automaton Once the canonical LR(1) automaton and corresponding parsing tables have been constructed, the LR(1) parser performs deterministic syntax analysis by simulating a left-to-right scan of the input, using a stack-based mechanism.

At the core of the parsing process are two tables derived from the automaton:

ACTION table maps each parser state and terminal symbol (including the end-of-input symbol \$) to one of four possible actions:

- *Shift* s_j: Advance to state j by consuming the current input symbol.
- *Reduce* r_k: Apply production rule $A \to \beta$ and reduce the right-hand side.
- *Accept*: Recognize the input as belonging to the language, which occurs only when the parser encounters the item $[S' \to S\cdot, \$]$.
- *Error*: No valid action exists for the current state and input; a syntax error is reported.

GOTO table maps each state and non-terminal symbol to a successor state, and is only used after reductions to determine the next parser state following a non-terminal transition.

Stack-based Parsing Mechanism The parser maintains a stack $\mathcal{S}$, initially containing only the start state s_0. The input string $w = a_1 a_2 \ldots a_n$ is augmented with the end-of-input marker \$, and a pointer is maintained to the current input symbol.

Each entry in the stack alternates between a grammar symbol and a state, *i.e.*,
$$\mathcal{S} = [X_0, s_0, X_1, s_1, \ldots, X_k, s_k]$$

The parsing loop proceeds as follows:

1. Let s_k be the state at the top of the stack, and let a be the current input symbol.
2. Consult the ACTION table at entry ACTION$[s_k, a]$:
   - *If it specifies Shift* s_j: Push the input symbol a and state s_j onto the stack, and advance the input pointer.
   - *If it specifies Reduce by production* $A \to \beta$: Pop $2|\beta|$ entries from the stack (removing both symbols and states), exposing the new top state s'. Push A, and then consult GOTO$[s', A] = s''$ to push the new state s''.
   - *If it specifies Accept*: The parser halts and returns success, confirming that the input belongs to the language defined by the grammar.
   - *If no action is defined (Error)*: The parser halts and reports a syntax error.
3. Repeat the above steps until either an Accept or Error action is encountered.

A.3 Definition of Pushdown Automata and Deterministic Pushdown Automata

A *pushdown automaton* (PDA) is a computational model that extends a finite automaton with a stack memory. Formally, a (nondeterministic) PDA is defined as a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where:

- Q is a finite set of states,
- Σ is a finite input alphabet,
- Γ is a finite stack alphabet,
- $\delta : Q \times (\Sigma \cup \epsilon) \times \Gamma; \to; \mathcal{P}(Q \times \Gamma^)$ is the transition function,
- $q_0 \in Q$ is the start state,
- $Z_0 \in \Gamma$ is the initial stack symbol,
- $F \subseteq Q$ is the set of accepting (final) states.

Here ϵ denotes the empty string and $\mathcal{P}(X)$ is the powerset of X. Intuitively, the PDA reads one symbol at a time (or makes ϵ-moves without consuming input), and at each step it may push or pop symbols on the stack. A transition of the form $\delta(q, a, A) \ni (p, \gamma)$ means that if the automaton is in state q, sees input symbol a (or $a = \epsilon$ for an ϵ-move), and A is the top stack symbol, then it can move to state p, pop A, and push the string $\gamma \in \Gamma$ onto the stack (with the leftmost character of γ becoming the new top). A string is accepted if the PDA can consume the entire input and reach a configuration in which the current state is in F (accepting), regardless of the remaining stack content. (Alternatively, acceptance by empty stack can be used; for nondeterministic PDAs these two acceptance modes yield the same class of languages.)

A *deterministic pushdown automaton* (DPDA) is a special kind of PDA with restrictions that eliminate nondeterminism. Formally, a DPDA is defined by the same 7-tuple structure, but its transition function δ must satisfy determinism conditions: for any state $q \in Q$ and stack symbol $A \in \Gamma$, at most one of the following can occur:

- There is at most one input symbol $a \in \Sigma$ such that $\delta(q, a, A)$ is nonempty (so for each fixed (q, A), there cannot be two different input symbols leading to transitions).
- If $\delta(q, \epsilon, A)$ is nonempty (an ϵ-move is available), then $\delta(q, a, A)$ must be empty for every $a \in \Sigma$ (so that no input-consuming move competes with an ϵ-move on the same (q, A)).

Informally, in a DPDA, the next move is uniquely determined by the current state, the current input symbol (or ϵ), and the top of the stack. Equivalently, for each (q, A) there is at most one transition available in total, and it cannot be both an ϵ-move and a non-ϵ-move simultaneously. A DPDA typically accepts by reaching an accepting state in F after consuming all input. A language is called *deterministic context-free* (a DCFL) if it is recognized by some DPDA; otherwise it may require a nondeterministic PDA.

A.4 Language Classes: CFL vs DCFL

The class of languages recognized by PDAs (nondeterministic) is exactly the class of *context-free languages* (CFLs). By contrast, the class of languages recognized by DPDAs is the class of *deterministic context-free languages* (DCFLs). It is known that DCFLs are a strict subset of CFLs: there are context-free languages that no DPDA can recognize. In general, every DPDA is also a PDA (so DCFL $\subseteq$ CFL), but many CFLs require nondeterminism.

Some key differences and properties include:

- **Expressive power:** Every DCFL is context-free, but there are CFLs that are not deterministic. For example, the language of all palindromes $ww^R : w \in a, b^*$ is context-free but not deterministic context-free (no DPDA can decide the midpoint of the string to switch from pushing to popping).

- **Unambiguity:** Every DCFL has an unambiguous context-free grammar and a unique leftmost derivation for each string. In contrast, CFLs in general may be ambiguous or inherently nondeterministic. (In fact, if a context-free language has a deterministic PDA, it admits no ambiguity in parsing.)

- **Closure properties:** DCFLs enjoy stronger closure than general CFLs. For instance, DCFLs are closed under complementation (by converting final-state acceptance to empty-stack acceptance and flipping accepting conditions), whereas CFLs are not closed under complement in general. Also, DCFLs are closed under intersection with regular languages. In contrast, CFLs are not closed under complement or intersection in general.

- **Parsing and complexity:** Deterministic PDAs can be executed in linear time, and they correspond to deterministic parsing algorithms (such as LL(1) or LR(1) parsers for programming languages). Nondeterministic PDAs also run in linear time (in theory) but require nondeterminism or backtracking to decide moves.

A.5 Transition Function and Determinism Conditions

The transition function δ of a PDA (resp. DPDA) encodes its moves. Recall $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$. If $(p, \gamma) \in \delta(q, a, A)$, this means the PDA can move from configuration $(q, aw, \dots)$ to $(p, w, \dots)$ by consuming a (unless $a = \epsilon$) and replacing the stack top A with the string γ. Concretely, if the current configuration is $(q, av, \alpha A)$ (state q, remaining input av, stack αA with A at top), then after the transition it goes to $(p, v, \alpha\gamma)$ (state p, remaining input v, stack $\alpha\gamma$). The string γ may be empty (denoted ϵ), which corresponds to simply popping A without pushing anything.

For a nondeterministic PDA, there may be multiple choices (p, γ) in the set $\delta(q, a, A)$, reflecting different possible moves. A DPDA imposes the restriction that these choices must be essentially unique. In particular:

- For each state q and stack symbol A, and for each input symbol $a \in \Sigma$, there is at most one pair $(p, \gamma) \in \delta(q, a, A)$. That is, the PDA cannot have two different transitions that read the same input a in the same state q with the same stack top A.

- Moreover, if $\delta(q, \epsilon, A)$ is nonempty (*i.e.*, an ϵ-move is available from (q, A)), then $\delta(q, a, A)$ must be empty for every $a \in \Sigma$. Equivalently, from any configuration (q, A), the machine cannot both use an ϵ-move and a non-ϵ-move; at most one type of move is allowed.

These conditions ensure that at most one transition is available in any situation, making the automaton deterministic. In practice, a DPDA's transition function is often given as a function rather than a relation, since there is at most one output move for each (q, a, A) (or (q, ϵ, A)).