LR Parsing of Permutation Phrases

Jana Kostičová[1]

[1]Faculty of Mathematics, Physics and Informatics, Comenius University, Mlynská dolina, Bratislava, Slovakia

Abstract

This paper presents an efficient method for LR parsing of permutation phrases. In practical cases, the proposed algorithm constructs an LR(0) automaton that requires significantly fewer states to process a permutation phrase compared to the standard construction. For most real-world grammars, the number of states required to process a permutation phrase of length n is typically reduced from $\Omega(n!)$ to $O(2^n)$, resulting in a much more compact parsing table. The state reduction increases with longer permutation phrases and a higher number of permutation phrases within the right-hand side of a rule. We demonstrate the effectiveness of this method through its application to parsing a JSON document.

Keywords

permutation phrase, LR parsing, state complexity, unordered content, JSON

1. Introduction

Several of today's languages allow for constructs consisting of unordered content, meaning that any permutation of child-subconstructs is allowed. Examples of such languages include Java, Haskell, XML, and JSON (JavaScript Object Notation) [1, 2]. JSON, an extremely popular tree format for data storage and transmission, is perhaps the most prominent example, as JSON objects always consist of zero or more unordered members. The structure of JSON data can be constrained by a schema, most commonly using JSON Schema [3]. A large part of such schema can be expressed using a context-free grammar (CFG) [4].

EBNF notation for CFGs requires all permutation options to be enumerated on the right-hand side of the rule that results in $n!$ grammar rules. For example, an unordered content over the symbols A, B, C yields $3! = 6$ rules:

$$S \to ABC|ACB|BAC|BCA|CAB|CBA$$

Cameron, in his work [5], proposed a shorthand notation for expressing unordered content, called a permutation phrase:

$$X \to \langle\!\langle\, A \,\|\, B \,\|\, C \,\rangle\!\rangle$$

Using this notation does not affect the expressiveness of CFGs, nor does it change the language generated by the grammar. However, it makes the language specification significantly more concise and easier to understand. Consequently, it is desirable to adapt common parsing algorithms to accept permutation phrases in the input grammar and to process them more efficiently than the original algorithms, which handled $n!$ rules.

The main contribution of this paper is a modification to LR parsing algorithms that enables efficient parsing of permutation phrases. The running time remains unaffected, and in practical cases, the number of states in the LR(0) / LR(1) automaton, as well as the size of the resulting parsing table, are significantly reduced compared to the original algorithm. This state reduction is achieved by changing the semantics of LR(0) items: instead of tracking the exact sequence of symbols already seen and expected, we only track the set of symbols seen and expected for permutation phrases. In the standard algorithm, the number of states for processing a permutation phrase of length n is computed as the sum of all k-permutations of n, $0 < k \leq n$. In the modified algorithm we only need to compute all k-combinations of n, $0 < k \leq n$.

ITAT'25: Information Technologies – Aplications and Theory, September 26–30, 2025, Telgárt, Slovakia

kosticova@dcs.fmph.uniba.sk (J. Kostičová)

Related work. To the best of our knowledge, ours it the first approach to efficient LR parsing of permutation phrases. There are few works that extend top-down parsing methods for this purpose: In [5], a modification to the LL parser is presented that keeps the $O(n)$ running time. In [6], a way how to extend a parser combinator library is proposed. An XML parser presented in [7] uses a two-stack pushdown automaton to parse XML documents against an LL(1) grammar with permutation phrases.

Algorithms for minimizing deterministic finite automata (DFA) [8, 9, 10] could be used to reduce the states of LR(0) automaton. However they cannot be applied directly since minimizing an LR(0) automaton is different from minimizing a general DFA – the content of the states must be taken into account to ensure proper shift/reduce actions. In addition, an extra step in the generation of the parsing table would be needed.

2. LR parsing

This section provides a brief informal overview of parsing, with a particular emphasis on LR parsing. We assume the reader is familiar with the concepts of context-free grammars and finite automata. We follow [11], where a formal treatment of these concepts can also be found.

By parsing, we mean recognizing the structure of a computer program or an instance of another type of language under consideration. This structure is typically described by a CFG, as CFGs can capture most of the syntactical constructs of common programming languages. During parsing, the goals are to decide the membership problem (i.e., whether a given string belongs to the language generated by the given CFG) and to construct the derivation, often represented as a derivation tree.

There exists a general algorithm for membership problem: the Cocke-Younger-Kasami (CYK) algorithm. However it has a time complexity of $O(n^3)$ and is therefore not convenient for real-world use cases.

Two methods are commonly used to achieve parsing in linear time: LL parsing and LR parsing. LL parsing is based on top-down approach meaning that it constructs the derivation tree from the root to the leaves. In contrast, LR parsing uses a bottom-up approach, constructing the derivation tree from the leaves to the root.

Both methods work only for restricted subsets of CFGs. These are referred to as LL(k) grammars and LR(k) grammars, respectively, where k denotes the length of lookahead – the number of symbols the parser examines ahead in the input string. The class of LR(k) grammars is a superset of the class of LL(k) grammars. This means that LR parsers are in general more powerful than LL parsers, and many widely used compilers and compiler generators are based on the LR parsing or one of its variants.

In this work, we will focus in more detail on LR parsing. As mentioned earlier, the derivation tree is constructed bottom-up; specifically, a right-most derivation in reverse is produced. The algorithm reads the input string and attempts to match the right-hand side (RHS) of a CFG rule. If such an RHS is found, it is reduced to the nonterminal on the left-hand side (LHS) of the corresponding rule. In this way, the algorithm derives the second-to-last sentential form of the right-most derivation. It then repeats this process using the new sentential form as input. The algorithm succeeds when the entire input string is reduced to the start symbol of the grammar. Failure is reported if no valid reduction can be made, or if a conflict arises – that is, either multiple reductions are possible at a given point (reduce/reduce conflict), or the parser cannot decide whether to reduce or to read the next input symbol (shift/reduce conflict).

The simplest of the LR-based parsers, called the SLR parser, is based on a finite state machine known as the LR(0) automaton. This automaton can be constructed automatically for a given CFG, hardcoding the CFG's rules into its states, and it handles exactly the logic described above. The states of an LR(0) automaton consist of LR(0) items, which are CFG rules with a dot inserted somewhere in their right-hand side (RHS). The dot divides the RHS into two parts – each possibly empty. In all LR parsing algorithms, the dot serves as a marker: the portion before the dot represents what part of the rule has already been recognized in the input, while the portion after the dot indicates what is yet to be recognized.

To achieve a deterministic LR(0) automaton, its states are defined as sets of LR(0) items – i.e., multiple rules may be in progress simultaneously. A reduction is performed when the automaton reaches a state

that contains an LR(0) item with the dot at the end, meaning the RHS of the corresponding rule has been fully recognized. Otherwise, the next symbol is read from the input string – this is known as the shift action.

3. Permutation phrases in context-free grammars

In this section we introduce necessary definitions related to extending context free grammars with permutation phrases. We follow the concept of permutation phrases that has been described informally in [5]. Without loss of generality, we assume that the CFG under consideration does not contain unreachable or non-generating nonterminals.

The RHS of a CFG rule is a sequence of grammar symbols called *a simple phrase*. Let us consider Σ to be an alphabet of both terminals and nonterminals of a CFG, then we refer to the set of simple phrases over Σ as $\Pi_s(\Sigma)$, i.e., $\Pi_s(\Sigma) = \Sigma^*$.

A *permutation phrase* over a set of non-empty simple phrases $\{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ is denoted by

$$\langle\!\langle\, \sigma_1 \parallel \sigma_2 \parallel \ \ldots\ \parallel \sigma_n \,\rangle\!\rangle,$$

where $n > 0$. We consider a permutation phrase to be only a specific notation for a set of non-empty elements, with its semantics explicitly defined by the $expand$ function. Let π be the permutation phrase over the set $\{A, B, C\}$, then $expand(\pi)$ returns all concatenated permutation options:

$$\pi = \langle\!\langle\, A \parallel B \parallel C \,\rangle\!\rangle \text{ then}$$

$$expand(\pi) = \{ABC, ACB, BAC, BCA, CAB, CBA\}.$$

We denote the set of all permutation phrases over simple phrases from $\Pi_s(\Sigma)$ by $\Pi_p(\Sigma)$. We use the following notations for permutation phrases in the rest of this paper:

- $|\pi|$ - size,
- $\pi \cup \pi'$ - union,
- $\pi \setminus \pi'$ - subtraction,
- $\sigma \in \pi$ - membership,
- $\{\pi_1, \pi_2\}$ - a partition of π into two subsets.

The permutation phrases containing the same elements are considered equal.

Permutation phrases can be integrated into the RHSs of CFG rules as a shorthand notation for unordered content. Then each rule with a permutation phrase replaces $n!$ enumerated rules. Let r be of the form

$$r : X \to Y\langle\!\langle\, A \parallel B \parallel CD \,\rangle\!\rangle.$$

Then we refer to the set of equivalent enumerated rules as $expand(r)$:

$$\begin{aligned} expand(r) \quad = \{ \quad & X \to YABCD, X \to YACDB, X \to YBACD, \\ & X \to YBCDA, X \to YCDAB, X \to YCDBA \ \} \end{aligned}$$

Definition 1. *Let Σ be an alphabet. The set $\Pi(\Sigma)$ of grammatical phrases with intergated permutation phrases over Σ and the expand function capturing their semantics are defined as follows:*

1. $\sigma \in \Pi_s(\Sigma)$ *then*
   - $\sigma \in \Pi(\Sigma)$,
   - $expand(\sigma) = \{\sigma\}$,
2. $\pi \in \Pi_p(\Sigma), \pi = \langle\!\langle \sigma_1 \parallel \sigma_2 \parallel \ \ldots\ \parallel \sigma_n \rangle\!\rangle, n > 0$ *then*
   - $\pi \in \Pi(\Sigma)$,
   - $expand(\pi) = \{\sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_n} \mid (i_1, \ldots, i_n) \in Perm(n)^{[1]}\}$,

[1]The set of all permutations of $\{1, \ldots, n\}$.

3. $\omega_1, \omega_2 \in \Pi(\Sigma)$ *then*
   - $\omega_1\omega_2 \in \Pi(\Sigma)$,
   - $expand(\omega_1\omega_2) = expand(\omega_1)\, expand(\omega_2)$[2].

Now we can define CFG with permutation phrases and its expanded grammar:

Definition 2. *A* context-free grammar with permutation phrases (CFGP) *is a 4-tuple $G = (N, T, P, S)$ where N is a set of nonterminals, T is a set of terminals, S is the initial nonterminal, and P is a finite set of rules $P \subseteq N \times \Pi(N \cup T)$. The* expanded grammar *of G is a CFG $G_e = (N, T, P_e, S)$ such that*

$$P_e = \bigcup_{r \in P} expand(r),$$

where $expand(X \to \omega) = \{X \to w \mid w \in expand(\omega)\}$.

Note that each CFG G is also a CFGP and in that case $G_e = G$. We refer to the rules of CFGP that contain permutation phrases as *permutation rules* and we use the shorthand notation $\Pi(G)$ for $\Pi(N \cup T)$. In the rest of this paper, we consider the following grammars (unless stated otherwise):

- $G = (N, T, P, S)$ is a CFGP,
- $G_e = (N, T, P_e, S)$ is the expanded CFG for G.

In what follows, we restrict our attention to permutation phrases that consist of symbols only. Possible approaches to overcome this limitation are discussed in Section 10.

4. LR(0) items of permutation rules

In this section we modify the definition of LR(0) items (shortly items) to cover also permutation rules. We first define the set of *item phrases*, i.e., the possible RHSs of the items. To distinguish which level of a given RHS is currently being recognized – whether the top level or a nested permutation phrase – we use dots annotated with superscripts (1) and (2), respectively.

Definition 3. *Given an alphabet Σ and $\omega \in \Pi(\Sigma)$, the set $IP(\omega)$ of item phrases of ω is defined as follows:*

1. $\omega \in \Pi_s(\Sigma)$ *is a simple phrase then*
$$IP(\omega) = \{\sigma_1 \cdot^{(1)} \sigma_2 \mid \sigma_1\sigma_2 = \omega\},$$
2. $\omega \in \Pi_p(\Sigma)$ *is a permutation phrase then*
$$IP(\omega) \;=\; \{\cdot^{(1)}\pi, \pi\cdot^{(1)}\} \cup \{\pi_1 \cdot^{(2)} \pi_2 \mid \{\pi_1, \pi_2\} \textit{ is a partition of } \pi\},$$
3. $\omega = \omega_1\omega_2, \omega_1, \omega_2 \in \Pi(\Sigma)$ *is a concatenation of phrases then*
$$IP(\omega) \;=\; \{\omega_1'\omega_2 \mid \omega_1' \in IP(\omega_1)\} \cup \{\omega_1\omega_2' \mid \omega_2' \in IP(\omega_2)\}.$$

Definition 4. *Let $r \in P$ be of the form: $X \to \omega$. The set of LR(0) items of r, denoted by $I(r)$, is defined by*

$$I(r) = \{[X \to \omega'] \mid \omega' \in IP(\omega)\}.$$

We extended the definition with the items of permutation rules. An item of the form $[X \to \pi_1 \cdot^{(2)} \pi_2]$, where $\{\pi_1, \pi_2\}$ is a partition of some permutation phrase π, indicates that the elements of π_1 have already been seen in some order, while the elements in π_2 are expected to be seen in some order. It means that the items for permutation phrases do not care about the exact order of the elements. The dot is marked by the superscript (2) meaning that the content of a permutation phrase is begin processed. On the other hand, the dot marked by the superscript (1) represents processing the RHS outside the permutation phrases and it has the same semantics as the dot in the original LR(0) items.

The items containing a permutation phrase are referred to as *permutation items*. We define the set of all items for a given CFGP as the union of items of its rules:

Definition 5. *The set of items for grammar G is defined by $I(G) = \bigcup_{r \in P} I(r)$.*

[2]The concatenation of sets $expand(\omega_1)$ and $expand(\omega_2)$.

5. Modified algorithm for generating LR(0) automaton

We first define the $head$ function which, for a given phrase ω of G, returns the set of grammar symbols that appear at the beginning of its expansions.

Definition 6. *The function* $head : \Pi(G) \to 2^{(N \cup T)}$ *is defined by*

- $\omega = \varepsilon$ *then* $head(\omega) = \emptyset$,
- $\omega = Y\omega'$, *where* $Y \in (N \cup T)$, *then* $head(\omega) = \{Y\}$,
- $\omega = \langle\!\langle \sigma_1 \parallel \sigma_2 \parallel \ldots \parallel \sigma_n \rangle\!\rangle\, \omega'$ *then* $head(\omega) = \{head(\sigma_1), head(\sigma_2), \ldots, head(\sigma_n)\}$.

Let $i \in I(G)$ *be an item of the form* $[X \to \alpha \cdot^{(j)} \beta]$, $j \in \{1, 2\}$, *then* $head$ *function for* i *is defined by:*

$$head(i) = head(\beta).$$

For a given item i and a grammar symbol Y, the partial function $step$ returns the item that results from the movement of the dot within the item i on symbol Y. The function is defined only if such movement is possible, i.e., $Y \in head(i)$.

Definition 7. *Let* $i \in I(G)$ *such that* $Y \in head(i)$. *Then the partial function* $step : I(G) \times (N \cup T) \to I(G)$ *is defined as follows*[3]*:*

1. *Matching the first level (a concatenation of phrases):*

   $i = [X \to \omega_1 \cdot^{(1)} \theta\omega_2]$, $\theta \in N \cup T \cup \Pi_p(G)$ *then*

   a) *if* $\theta = Y$*:*
   $$step(i, Y) = [X \to \omega_1 Y \cdot^{(1)} \omega_2]$$

   b) *if* $\theta \in \Pi_p(G), Y \in \theta$*:*
   $$step(i, Y) = [X \to \omega_1 \boxed{\langle\!\langle Y \rangle\!\rangle \cdot^{(2)} (\theta \setminus \{Y\})} \omega_2]$$

2. *Matching the second level (the content of a permutation phrase):*

   $i = [X \to \omega_1 \boxed{\pi_1 \cdot^{(2)} \pi_2} \omega_2]$, $\pi_1, \pi_2 \in \Pi_p(G)$ *then*

   a) *if* $\pi_2 = \langle\!\langle Y \rangle\!\rangle$*:*
   $$step(i, Y) = [X \to \omega_1(\pi_1 \cup \{Y\}) \cdot^{(1)} \omega_2]$$

   b) *if* $|\pi_2| \geq 2$ *and* $Y \in \pi_2$*:*
   $$step(i, Y) = [X \to \omega_1 \boxed{(\pi_1 \cup \{Y\}) \cdot^{(2)} (\pi_2 \setminus \{Y\})} \omega_2]$$

Example 1. *Let us consider an item with the dot at the beginning of its RHS, indicating that the corresponding rule is just starting to be recognized:*

$$i_0 = [X \to \cdot^{(1)} A\langle\!\langle\, B \parallel C \,\rangle\!\rangle\, D].$$

Using the notation $i \to^{step\,Y} j$ *for* $step(i, Y) = j$*, we get the following sequence of successive application of the step function:*

$$\begin{array}{lll} i_0 & \to^{step\,A} & [X \to A \cdot^{(1)} \langle\!\langle\, B \parallel C \,\rangle\!\rangle\, D] \\ & \to^{step\,C} & [X \to A\, \langle\!\langle\, C \,\rangle\!\rangle \cdot^{(2)} \langle\!\langle\, B \rangle\!\rangle D] \\ & \to^{step\,B} & [X \to A\, \langle\!\langle\, B \parallel C \,\rangle\!\rangle \cdot^{(1)} D] \\ & \to^{step\,D} & [X \to A\, \langle\!\langle\, B \parallel C \,\rangle\!\rangle\, D \cdot^{(1)}] \end{array}$$

Note that the step function preserves the rule being recognized, meaning that both the original item and the resulting item belong to the set of items $I(r)$ for the same rule r.

[3] For better understanding, the processing of the content of a permutation phrase is marked by a box.

Proposition 1. *Let $r \in P$ and $i \in I(r)$ then $(step(i, Y) = j) \Rightarrow j \in I(r)$.*

Actually, the superscripts of the dot help to preserve the rule being processed as shown in the following example.

Example 2. *If we do not mark the dot to determine the level being processed, then the item*

$$i = [X \to \langle\langle\, A \parallel B \,\rangle\rangle \cdot \langle\langle\, C \parallel D \,\rangle\rangle]$$

can originate from two different rules r_1, r_2 and $step(i, D)$ can result in two different options j_1, j_2:

$$\begin{array}{llll} r_1: & X \to \langle\langle\, A \parallel B \,\rangle\rangle\langle\langle\, C \parallel D \,\rangle\rangle, & j_1: & [X \to \langle\langle\, A \parallel B \,\rangle\rangle\langle\langle\, C \,\rangle\rangle \cdot \langle\langle\, D \,\rangle\rangle] \\ r_2: & X \to \langle\langle\, A \parallel B \parallel C \parallel D \,\rangle\rangle, & j_2: & [X \to \langle\langle\, A \parallel B \parallel C \,\rangle\rangle \cdot \langle\langle\, D \,\rangle\rangle] \end{array}$$

Marking the dot in i by the superscript (1) indicates that the top level is being processed - that corresponds to the rule r_1 and the resulting item j_1. Marking the dot by the superscript (2) indicates that the permutation phrase is being processed - that corresponds to the rule r_2 and the resulting item j_2.

We modify the standard algorithm for generating LR(0) automaton [11] as follows:

```
SetOfItems PERM_CLOSURE(I) {
    J = I;
    repeat
        for (each item [A → α ·(i) β] in J such that i ∈ {1, 2})
            for (each nonterminal B ∈ head(β))
                for (each production B → γ of G)
                    if ([B → ·(1)γ] not in J)
                        add [B → ·(1)γ] to J;
    until no more items are added to J on one round;
    return J;
}

SetOfItems PERM_GOTO(I, Y) {
    J = ∅;
    for (each item [X → α ·(i) β] of I such that Y ∈ head(β) and i ∈ {1, 2})
        add step([X → α ·(i) β], Y) to J;
    return PERM_CLOSURE(J);
}
```

The algorithm uses the generic functions $head$ and $step$ defined above, which can be applied to any phrase, including those with integrated permutation phrases. The algorithm for constructing LR(0) parsing table with shift/reduce actions remains unchanged. We define LR(0) automaton for CFGP G as follows:

Definition 8. *Let $G' = (N', T, P', S')$ be the augmented grammar of G such that $N' = N \cup \{S'\}$ and $P' = P \cup \{S' \to S\}$. Then the* LR(0) automaton for G *is a DFA $A = (\Sigma, Q, \delta, I_0, F)$ such that*

- $\Sigma = N \cup T$, $I_0 = \{[S' \to \cdot^{(1)} S]\}$,
- *Q and δ are constructed by* PERM_GOTO *function, they are extended by an error state $\emptyset$ and transitions from/to this error state in a standard way to make A complete.*

Example 3. *Let us consider processing a CFG rule*

$$X \to \langle\langle A \parallel B \parallel C \rangle\rangle.$$

The corresponding part of the LR(0) automaton has 8 states and the transitions among these states are depicted in Figure 1. The LR(0) automaton for the extended grammar would have 16 states, as it allows only exact sequences before the dot. Namely, states I_5, I_6 and I_7 would be duplicated for the different prefixes, and the single state I_8 would be split into 6 separate rules, corresponding to the 6 permutations.

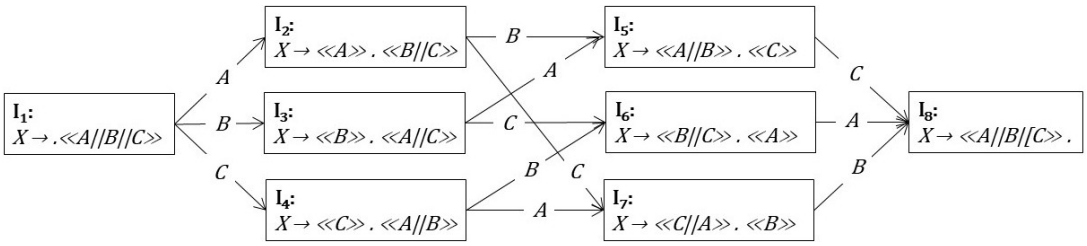

Figure 1: Processing of a permutation phrase

For simplicity, we assumed a single item per state in the example above. In general, there may be multiple items within a state, and some of them may even interfere with the permutation items depicted. Such a situation is discussed later in Section 6.

In addition to the grammars G, G_e defined before, we consider the following definitions in the rest of this paper (unless stated otherwise):

- $A = (\Sigma, Q, \delta, I_0, F)$ is the complete LR(0) automaton for the grammar G allowing permutation rules constructed by our modified algorithm,
- $A_e = (\Sigma, Q_e, \delta_e, I_{0e}, F_e)$ is the complete LR(0) automaton for the extended grammar G_e constructed by the standard algorithm [11].

6. Map function

In this section, we define an 1:N mapping between the states of A and A_e. We later use this mapping to prove the correctness of our modified algorithm for constructing the LR(0) automaton, as well as to carry out a complexity analysis.

We first introduce some supplementary concepts. In the LR(0) automaton, there is a close relationship between the input strings leading to a state I (called input paths) and before-the-dot parts of the items within I. To make the set of inputs paths for such a state I finite, we consider only those whose lengths are limited by the longest before-the-dot part among the items in I.

Definition 9. *Let $I \in Q$ and $max = max\{|\alpha| \mid [X \to \alpha \cdot \beta] \in I\}$. Then the function $paths : Q \to 2^{\Sigma^*}$ returns* input paths for I *and is defined by:*

$$paths(I) = \{w \mid 0 < |w| \leq max \text{ and there exists } I' \in Q \text{ such that } (I', w) \vdash_A^* (I, \varepsilon)\}.$$

Note that $paths(I)$ is empty only for the initial state I_0 and if w is in $paths(I)$ then also all suffixes of w (except the empty word) are in $paths(I)$. For the LR(0) automaton of a grammar with permutation rules, the following proposition captures the relation between $paths(I)$ and the before-the-dot parts of items in I.

Proposition 2. *Let $I \in Q$ and $i \in I$ be an item of the form $i : [X \to \alpha \cdot \beta]$. Then*

$$paths(I)/_{|\alpha|} \subseteq expand(\alpha)$$

where $paths(I)/_{|\alpha|}$ is a set of all suffixes of $paths(I)$ of length $|\alpha|$.

Note that if the grammar has no permutation rules, the equality holds:

$$paths(I)/_{|\alpha|} = expand(\alpha) = \{\alpha\}.$$

This means that any viable prefix leading to I has the before-the-dot part α as its exact suffix of length $|\alpha|$ and no other strings appear as suffixes of that length.

If permutation items are present, they may interfere with other items in such a way that some states are "split". Such a situation is depicted in Figure 2: there are two states, I_5 and I_6, both containing the

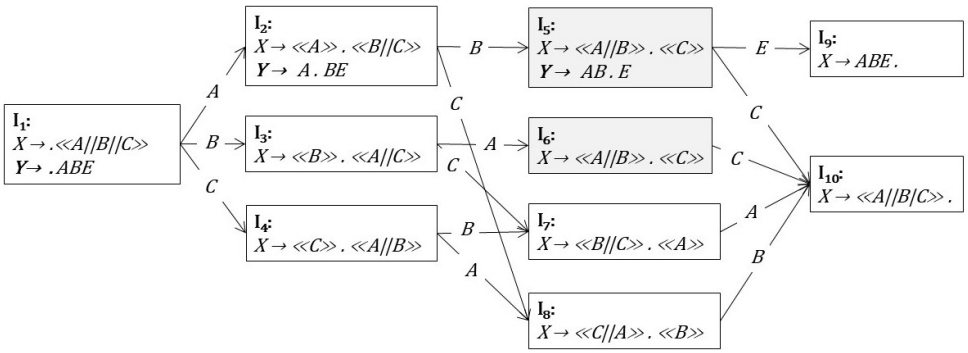

Figure 2: An example of rule interference

item $i = [X \to \langle\langle A \parallel B \rangle\rangle \cdot \langle\langle C \rangle\rangle]$. The splitting is caused by an interfering item $[Y \to AB \cdot E]$. Let us recall that in Figure 1, where the same rule is being processed but no interferences were assumed, there was only a single state containing i, namely I_5.

The PERM_GOTO function automatically handles rule interferences and generates as many states for A as are actually needed. The more interferences occur, the more states of A are generated, which results in a lower state reduction between A and A_e.

We say that a CFGP rule is independent when its items do not interfere with other items within the same state. Thus, independency is defined based on the equality between the sets $paths(I)/_{|\alpha|}$ and $expand(\alpha)$ for each rule item that appears in some $I \in Q$. This concept is not required for the definition of the mapping function, but it will be used later in the complexity analysis.

Definition 10. *Let $I \in Q$ and $i \in I$ be an item of the form $i : [X \to \alpha \cdot \beta]$. Then the item i is* independent *in I if and only if $paths(I)/_{|\alpha|} = expand(\alpha)$.*

Definition 11. *Let $r \in P$, r is* independent *if and only if for each $i \in I(r)$ and $I \in Q$:*

$$i \in I \text{ implies } i \text{ is independent in } I.$$

Now we can proceed with the definition of the map function itself. Intuitively, this 1:N mapping translates a state I of A (the LR(0) automaton for the CFGP) into a set of states of A_e (the LR(0) automaton for the expanded grammar) in such a way that each item containing a permutation phrase before the dot induces a separate state in A_e for each expansion of this permutation phrase. However, dependent rules must also be taken into account, as in such cases, some states in A may already be split (fully or partially). Both situations are depicted in Figure 3.

We first define a function map_{state} with two arguments: for an input state $I \in Q$ and an input string w, it returns a state $I_e \in Q_e$, which handles the processing of the input paths of $paths(I)$ that are suffixes of w. The value of $map(I, w)$ is undefined if no suffix of w is in $paths(I)$.

We use two auxilliary partial functions map_{phrase} and map_{item} that map phrases and items of G to phrases and items of G_e, respectively, with respect to w.

Definition 12. *The partial function $map_{phrase} : \Pi(G) \times \Sigma^* \to \Pi(G_e)$ is defined by:*

$$map_{phrase}(\alpha, w) = \alpha' \in (N \cup T)^* \text{ where } \alpha' \in expand(\alpha) \text{ and } \alpha' \text{ is a suffix of } w.$$

The partial function $map_{item} : I(G) \times \Sigma^* \to 2^{I(G_e)}$ is defined by:

$$map_{item}([X \to \alpha \cdot \beta], w) = \{[X \to \alpha' \cdot \beta'] \in I(G_e) \mid \alpha' = map_{phrase}(\alpha, w), \beta' \in expand(\beta)\}. \quad (1)$$

The function $map_{state} : Q \times \Sigma^* \to Q_e$ is defined by:

- *$I = \emptyset$ then $map(I, w) = \emptyset$ for any $w \in \Sigma^*$ (error state),*
- *$I \neq \emptyset$ and $w \neq w_1 w_2$ such that $w_2 \in paths(I)$ then $map(I, w) = \emptyset$,*

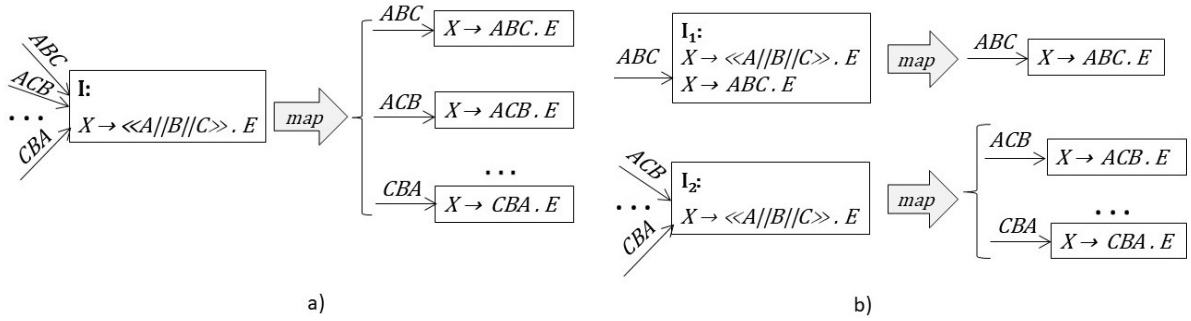

Figure 3: The mapping of a) an independent rule b) a dependent rule

- $I \neq \emptyset$ *and* $w = w_1 w_2$ *such that* $w_2 \in paths(I)$ *then* $map(I, w) = \bigcup_{i \in I} map_{item}(i, w)$.

Building on the preceding stepwise definitions, we now introduce the final map function. It maps a state I of A to the set of the A_e states that handle the processing of the paths in $paths(I)$ as shown in Figure 3. Note that only the before-the-dot parts induce multiple states in A_e. The after-the-dot parts containing permutation items are always expanded within the same state of A_e allowing recognition of any of the permutation options while processing the rest of the input (see (1)).

Definition 13. *The function* $map : Q \rightarrow 2^{Q_e}$ *is defined by:*

$$map(I) = \bigcup_{\substack{w \in \\ paths(I)}} map(I, w).$$

7. Correctness

Now we prove the key statements to show that A is correct: the states reached by A and A_e on the same input can be related by the map function and additionally, A and A_e return the same parser action. Note that both A and A_e perform m computation steps to process an input of length m since they are deterministic and complete.

Lemma 1. *Let* $w \in \Sigma^*$ *and* $(I_0, w) \vdash_A^{|w|} (J, \varepsilon), (I_{0e}, w) \vdash_{A_e}^{|w|} (J_e, \varepsilon)$. *Then* $J_e = map(J, w)$.

Proof. We give a proof by induction on $|w|$. The base case $|w| = 0$ trivially holds. Let us assume the statement holds for $n = k$ and let us have the following computations on the input $|w| = k + 1$, where $w = w'Y, Y \in \Sigma$:

$$\begin{array}{llll} (I_0, w'Y) & \vdash_A^k & (I, Y) & \vdash_A (J, \varepsilon) \text{ and} \\ (I_{0e}, w'Y) & \vdash_{A_e}^k & (I_e, Y) & \vdash_{A_e} (J_e, \varepsilon). \end{array} \tag{2}$$

Then it also holds:

$$(I_0, w') \vdash_A^k (I, \varepsilon) \text{ and } (I_{0e}, w') \vdash_{A_e}^k (I_e, \varepsilon).$$

Based on the induction hypothesis $I_e = map(I, w)$. We need to prove $J_e = map(J, w'Y)$. It is sufficient to prove that mapping holds for kernel items[4] - $kernel(J_e) = map(kernel(J), w'Y)$ – as that implies that the mapping holds for non-kernel items as well and thus $J_e = map(J, w'Y)$.
We define the subsets $I_Y \subseteq I, I_{eY} \subseteq I_e$ that participate in the computation step of A and A_e, respectively, on the symbol Y:

$$\begin{array}{lcl} I_Y & = & \{i \in I, Y \in head(i)\}, \\ I_{eY} & = & \{i_e \in I_e, Y \in head(i_e)\}. \end{array} \tag{3}$$

[4]Kernel items are those that do not have the dot at the beginning of the RHS. The items with a dot at the beginning of RHS are non-kernel items.

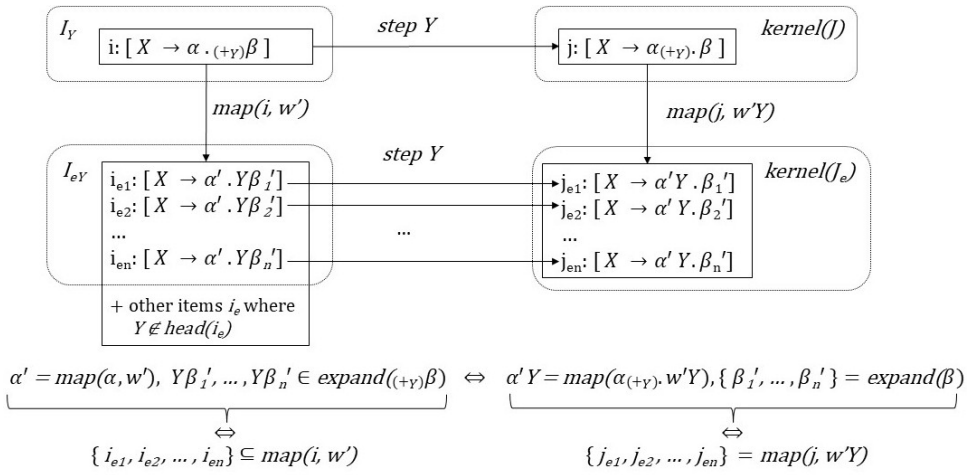

Figure 4: The items participating in the last computation step of A and A_e (on the symbol Y) and their mutual relationships

Based on the definition of δ and δ_e functions it holds

$$\begin{array}{lcl} \delta(I, Y) & = & \delta(I_Y, Y) = J, \\ \delta_e(I_e, Y) & = & \delta_e(I_{eY}, Y) = J_e. \end{array} \tag{4}$$

If $I_Y = \emptyset$, then $J = J_e = \emptyset$ and the statement clearly holds. Assume $I_Y \neq \emptyset$. The situation is depicted in Figure 4. We use the following shorthand notations:

- $\alpha_{(+Y)}$ to refer to the phrase resulting from adding Y to the end of α:
  - If $\alpha = \omega\pi$ where π is a permutation phrase then $\alpha_{(+Y)} = \omega(\pi \cup \{Y\})$.
  - If $\alpha = \omega A$ where $A \in N \cup T$ then $\alpha_{(+Y)} = \omega AY$.
- ${}_{(+Y)}\beta$ to refer to the phrase resulting from adding Y to the beginning of β:
  - If $\beta = \pi\omega$ where π is a permutation phrase then ${}_{(+Y)}\beta = (\pi \cup \{Y\})\omega$.
  - If $\beta = A\omega$ where $A \in N \cup T$ then ${}_{(+Y)}\beta = YA\omega$.

It is easy to see that

$$\begin{array}{rcl} j_e \in map(kernel(J), w'Y) & \Rightarrow & \exists j \in kernel(J) : j_e \in map(j, w'Y) \Rightarrow \\ & \Rightarrow & \exists i \in I_Y : step(i, Y) = j \Rightarrow \\ & \Rightarrow & \exists i_e \in I_{eY} : i_e \in map(i, w'), step(i_e, Y) = j_e \Rightarrow \\ & \Rightarrow & j_e \in kernel(J_e), \end{array}$$

$$\begin{array}{rcl} j_e \in kernel(J_e) & \Rightarrow & \exists i_e \in I_{eY} : step(i_e, Y) = j_e \Rightarrow \\ & \Rightarrow & \exists i \in I_Y : map(i, w') = i_e \Rightarrow \\ & \Rightarrow & \exists j \in J : j = step(i, Y), map(j, w'Y) = j_e \Rightarrow \\ & \Rightarrow & j_e \in map(kernel(J), w'Y). \end{array}$$

□

Theorem 1. *Let $w \in \Sigma^*$ and $Y \in \Sigma$ and $(I_0, w) \vdash_A^* (J, \varepsilon), (I_{0e}, w) \vdash_{A_e}^* (J_e, \varepsilon)$. Then*

$$action(J, Y) = action(J_e, Y) \text{ or } J = J_e = \emptyset.$$

Proof. Based on Lemma 1 we get $J_e = map(J, w)$. Then it holds $J = \emptyset \Leftrightarrow J_e = \emptyset$. Let assume $J, J_e \neq \emptyset$. Based on the definition of map function, an item $j \in J$ has a transition on Y if and only if at least one of the items $j_e \in map(j, w)$ has a transition on Y. That implies

$$(shift) \in action(J, Y) \iff (shift) \in action(J_e, Y).$$

At the same time, an item $j \in J$ is of the form $[X \to \alpha \cdot^{(1)}]$ if and only if $map(j, w) = \{j_e\}$ and j_e is of the form $[X \to \sigma \cdot]$ where $\sigma = map(\alpha, w)$. This means

$$(reduce\, X \to \alpha) \in action(J, Y) \iff (reduce\, X \to \sigma) \in action(J_e, Y) \text{ where } \sigma \in expand(\alpha).$$

□

8. State complexity

In this section, we analyze the difference between the number of states needed to process a permutation phrase in A and to process all corresponding permutation options in A_e. We also discuss the differences in processing simple phrases in permutation rules. The greatest state reduction is achieved when a rule is independent, meaning that processing permutation phrases on the rule's RHS does not interfere with other rules or other parts of the same rule.

Definition 14. *Let $r \in P$ be a rule of the form $r : X \to \omega$. Let $I_0 \in Q$ be a state of A that contains item $[X \to \cdot^{(1)} \omega]$. Then the* set of states processing the rule r in A starting from I_0 *is defined by $\mathscr{I}(r, I_0) = \bigcup_{k=0}^{|\omega|} \mathscr{I}_k(r, I_0)$ where*

1. $\mathscr{I}_0(r, I_0) = \{I_0\}$,
2. $\mathscr{I}_k(r, I_0) = \{I \mid I = \delta(I', Y)$ *where*
   - $I' \in \mathscr{I}_{k-1}(r, I_0)$ *and there exists* $i \in I' \cap I(r)$ *of the form* $[X \to \omega_1 \cdot^{(j)} \omega_2]$ *where* $j \in \{1, 2\}$, $|\omega_1| = k - 1$ *and* $Y \in head(\omega_2)\}$.

Note that the set $\mathscr{I}_k(r, I_0)$ contains all states reached from I_0 by processing the first k symbols of ω and the set $\bigcup_{k=i}^{j} \mathscr{I}_k(r, I_0)$ contains all states needed to process the subphrase of ω between the i-th and the j-th symbol. If we replace P, A, δ with P_e, A_e, δ_e, respectively, we get similar definition for the extended LR(0) automaton A_e.

Theorem 2. *Let $r \in P$ be an independent rule of the form:*

$$X \to \sigma_0 \pi_1 \sigma_1 \dots \sigma_{m-1} \pi_m \sigma_m$$

where each $\sigma_i \in \Pi_s(G)$ is a simple phrase and each $\pi_i \in \Pi_p(G)$ is a permutation phrase. Then the processing of σ_i/π_i in A/A_e requires the following number of states:

A/π_i : *at most* $2^{\lvert\pi_i\rvert}$ *states* $(\in O(2^{\lvert\pi_i\rvert}))$,	A_e/π_i : *at least* $M \sum_{k=1}^{\lvert\pi_i\rvert} P(\lvert\pi_i\rvert, k)$ *states* $(\in \Omega(\lvert\pi_i\rvert!))$,
A/σ_i : *at most* $\lvert\sigma_i\rvert$ *states,*	A_e/σ_i : *at least* $M\lvert\sigma_i\rvert$ *states.*

where $M = \prod_{j=1}^{i-1} |\pi_j|!$ is the multiplication factor and $P(|\pi_i|, k) = \frac{|\pi_i|!}{(|\pi_i|-k)!}$ is the number of all k-permutations of π_i.

Proof. Let us denote the part of the RHS of rule r before π_i by ω_1 and the part after it as ω_2; i.e., $r = X \to \omega_1 \pi_i \omega_2$.

Processing of π_i in A starts in a state $I \in \mathscr{I}_{|\omega_1|+1}(r, I_0)$ for some I_0 meaning that the part of the rule before π_i is processed between the states I_0 and I. The state I contains the item of the form $[X \to \omega_1 \cdot^{(1)} \pi_i \omega_2]$. Then A passes the states that contain the following items:

$$[X \to \omega_1 \pi_1 \cdot^{(2)} \pi_2 \omega_2], \{\pi_1, \pi_2\} \text{ is a partition of } \pi_i, \text{ and } [X \to \omega_1 \pi_i \cdot^{(1)} \omega_2] \tag{5}$$

where π_1 can be any of the k-combinations of π_i for $0 < k < |\pi_i|$. When we count all options for (5), we get the number of the states needed for processing π_i in r starting from I_0:

$$\left| \bigcup_{k=|\omega_1|+1}^{|\omega_1|+|\pi_i|} \mathscr{I}_k(r, I_0) \right| = \sum_{k=1}^{|\pi_i|} C(|\pi_i|, k) = 2^{|\pi_i|} - 1. \tag{6}$$

Let $I_{0e} \in map(I_0)$, and let A_e be in a state I_e such that $(I_{0e}, w_1) \vdash_{A_e} (I_e, \varepsilon)$ for some $w_1 \in expand(\omega_1)$. Based on the definition of the map function and Lemma 1, I_e contains all items of the form:

$$[X \to w_1 \cdot \sigma w_2] \text{ where } \sigma \in expand(\pi_i) \text{ and } w_2 \in expand(\omega_2).$$

While processing the phrase σ, A_e passes states that contain items of the form

$$[X \to w_1\sigma_1 \cdot \sigma_2 w_2] \text{ where } \sigma_1\sigma_2 \in expand(\pi_i), \sigma_1 \neq \varepsilon, w_2 \in expand(\omega_2) \tag{7}$$

where σ_1 can be any of the k-permutations of π_i for $0 < k \leq |\pi_i|$. When we count all the options for w_1, σ_1 and σ_2, we get the number of states needed for processing all expansions of π_i starting from the states of $map(I)$:

$$\left| \bigcup_{r_e} \bigcup_{k=|w_1|+1}^{|w_1|+|\sigma|} \mathscr{J}_k(r_e, I_{0e}) \right| = M \sum_{k=1}^{|\pi_i|} P(|\pi_i|, k) = M \sum_{k=1}^{|\pi_i|} \frac{|\pi_i|!}{(|\pi_i| - k)!} \geq M|\pi_i|! \tag{8}$$

where $r_e \in Q_e$ ranges over rules of the form $X \to w_1\sigma w_2$ with $w_1 \in expand(\omega_1)$, $\sigma \in expand(\pi_i)$, and $w_2 \in expand(\omega_2)$. The multiplication factor M represents the distinct choices of w_1 and equals the product of the numbers of permutation options for the permutation phrases in ω_1:

$$M \;=\; \prod_{j=1}^{i-1} |\pi_j|!.$$

Note that w_2 does not affect M, since it is the part of the rule that is processed later.

The statements for a simple phrase σ_i can be proved similarly. In this case the processing proceeds in the same way both in A and A_e; however, the multiplication factor is again be applied for A_e. □

We analyzed the state reduction for independent rules at the local (rule) level. With dependent rules, some states of A are split and in the worst case, the number of the states of A equals to the number of the states of A_e. It cannot be lower, as a state of A can be split into at most as many states as is the number of its input paths and that is exactly the number of A_e states. However, the real-world grammars typically contain no or just very few rule interferences.

At the global (grammar) level, more types of rule interferences may appear. For example, if two permutation phrases of different rules are processed in parallel (i.e., the same sequence of states of A is used), the corresponding reduction applies only once. On the other hand, if permutation phrases of different rules are processed one by one, the global multiplication factor – similar to the local one mentioned in Theorem 2 – applies as well.

8.1. JSON Example

We provide an example of JSON schema in the form of CFGP grammar and demonstrate the state reduction achieved by our modified algorithms. Consider the following CFGP grammar G that define the content of the complex objects and arrays (the rules are numbered):

$$\begin{array}{lcl}
catalog & \to & catalogItem\ (1), \\
 & | & catalogItem\ catalog\ (2) \\
catalogItem & \to & \langle\!\langle\ \texttt{id} \parallel \texttt{name} \parallel addresses\ \rangle\!\rangle\ (3) \\
addresses & \to & address\ (4) \\
 & | & address\ addresses\ (5) \\
address & \to & \langle\!\langle\ \texttt{addressId} \parallel \texttt{home} \parallel \texttt{street} \parallel \texttt{no} \parallel \texttt{city} \parallel \texttt{code}\ \rangle\!\rangle\ (6)
\end{array}$$

The right-hand sides of the rules 3 and 6 consist of a permutation phrases of length 3 and 6, respectively. It is easy to see that both rules are independent. When we construct the LR(0) automaton A for G and A_e for the expanded grammar of G_e, we obtain the following number of states needed for processing

the permutation rules[5] - note that the state reduction increases rapidly as the length of the permutation phrase grows:

$$A\text{/rule 3: at most } 2^3 = 8, \quad A_e\text{/rule 3: at least } \sum_{k=0}^{3} \frac{3!}{(3-k)!} = 16,$$
$$A\text{/rule 6: at most } 2^6 = 64, \quad A_e\text{/rule 6: at least } \sum_{k=0}^{6} \frac{6!}{(6-k)!} = 1975.$$

9. Construction of SLR / canonical LR / LALR parsing tables

We describe the modification to the standard algorithms for constructing SLR / canonical LR / LALR parsing tables [11] so that they can process CFGPs. Two functions are needed - $FIRST$ and $FOLLOW$ and we extend them to handle permutation rules:

Extension of the $FIRST$ function:

- $\omega = \langle\!\langle\, \sigma_1 \parallel \sigma_2 \parallel \ldots \parallel \sigma_n \,\rangle\!\rangle$ then $FIRST(\omega) = \bigcup_{0 \le i \le n} FIRST(\sigma_i)$,
- $\omega = \omega_1\omega_2, \omega_1, \omega_2 \in \Pi(G)$ then
  - if $\varepsilon \in FIRST(\omega_1)$ then $FIRST(\omega) = FIRST(\omega_1) \cup FIRST(\omega_2)$,
  - if $\varepsilon \notin FIRST(\omega_1)$ then $FIRST(\omega) = FIRST(\omega_1)$.

Extension of the FOLLOW function: Let $r : X \to \omega_1\pi\omega_2$ be a rule of G. If $Y \in \pi$ then

- for each $Y' \in \pi$, $Y' \neq Y$, add $FIRST(Y') \setminus \{\varepsilon\}$ to $FOLLOW(Y)$,
- add $FIRST(\omega_2) \setminus \{\varepsilon\}$ to $FOLLOW(Y)$,
- if $\omega_2 = \varepsilon$ or $\varepsilon \in FIRST(\omega_2)$ then add $FOLLOW(X)$ to $FOLLOW(Y)$.

We get LR(1) items by adding lookahead to the LR(0) items. The body of the repeat loop in the closure function for LR(1) items is modified as follows[6]:

for (each item $[A \to \alpha \cdot^{(i)} \beta, a]$ in J such that $i \in \{1, 2\}$)
  for (each nonterminal $B \in head(\beta)$)
    for (each production $B \to \gamma$ of G)
      for (each symbol b in $FIRST({}_{(-B)}\beta a)$)
        if ($[B \to \cdot^{(1)}\gamma, b]$ not in J)
          add $[B \to \cdot^{(1)}\gamma, b]$ to J;

Assume A and A_e are LR(1) automata for G and G_e, respectively, constructed using the PERM_CLOSURE function above. The map function for an LR(1) item and an input string w maps the LR(0) part of the item in the same way as for LR(0) items and does not manipulate the lookahead part. Let I be a state of A. Each of the mapped states I_e contains all expansions of the phrases that appear after the dots in I. When constructing parsing table for canonical LR parser, the states of A are split based on the lookahead only if the mapped states of A_e are also split, preserving the state reduction rate. Similarly, when merging states for an LALR parser, the state reduction rate remain unaffected.

10. Conclusion and future work

We presented a modification of LR parsing algorithms that, in practical cases, generates significantly smaller parsing tables. For independent rules, the number of states needed for processing a permutation phrase π of size n in LR(0) automaton is reduced from $\Omega(n!)$ to $O(2^n)$. The reduction in the number of states increases with the size of π as well as its placement within the right-hand side of a rule. The more permutation phrases appear before π, the higher the reduction. In addition to providing a more efficient

[5] We also included the item having the dot at the beginning.
[6] We use the notation ${}_{(-B)}\beta$ to denote the phrase obtained by removing B from the beginning of β.

approach for processing permutation phrases in existing languages, we hope that the findings of this work will also assist language designers in making informed decisions about incorporating permutation phrases into their specifications.

Our algorithm does not support nested simple phrases and optional elements within a permutation phrase. For nested phrases, another level of processing needs to be introduced and the $step$ function must be extended to handle that level. It is required that, within a permutation phrase, one nested simple phrase is not a prefix of another to avoid conflicts. Optional elements require the modification of the $head$ function and they cannot conflict with the set of symbols that can follow given permutation phrase. In both cases the limitations could be possibly avoided by parallel processing of more items. It would be beneficial to extend the algorithm to handle nested simple phrases and optional elements without limitations. Another possible direction for future work is to explore in detail the relationship between the number and type of rule interferences and the resulting reduction in states, as well as to analyze the global state reduction at the grammar level.

Declaration on Generative AI

During the preparation of this work, the author used ChatGPT-4 to check grammar, spelling, and improve sentence clarity. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] ECMA-404 The JSON data interchange syntax, 2nd Edition, ECMA, 2017. https://ecma-international.org/\publications-and-standards/standards/\ecma-404/.

[2] The JavaScript Object Notation (JSON) Data Interchange Format, Internet Engineering Task Force (IETF), 2017. https://datatracker.ietf.org/doc/html/\rfc8259.

[3] B. Hutton, C. Bormann, G. Normington, H. Andrews, JSON Schema: A Media Type for Describing JSON Documents, https://json-schema.org/draft/2020-12/json-schema-core, 2020. Internet-Draft, work in progress.

[4] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation (3rd ed.), Pearson, 2013.

[5] R. D. Cameron, Extending context-free grammars with permutation phrases, ACM Lett. Program. Lang. Syst. 2 (1993) 85–94.

[6] A. I. Baars, A. Löh, S. D. Swierstra, Parsing permutation phrases, J. Funct. Program. 14 (2004) 635–646.

[7] W. Zhang, R. Engelen, High-Performance XML Parsing and Validation with Permutation Phrase Grammar Parsers, 2008, pp. 286–294.

[8] J. Hopcroft, An n log n algorithm for minimizing states in a finite automaton, in: Z. Kohavi, A. Paz (Eds.), Theory of Machines and Computations, Academic Press, 1971, pp. 189–196.

[9] J. Brzozowski, Canonical regular expressions and minimal state graphs for definite events, Proc. Symposium of Mathematical Theory of Automata 12 (1962) 529–561.

[10] E. F. Moore, Gedanken-Experiments on Sequential Machines, in: C. Shannon, J. McCarthy (Eds.), Automata Studies, Princeton University Press, Princeton, NJ, 1956, pp. 129–153.

[11] A. V. Aho, S. Ravi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (1st ed.), Addison-Wesley, 1986.