# Towards a 2-Multiple Context-Free Grammar for the 3-dimensional Dyck Language

Konstantinos Kogkalidis and Orestis Melkonian

Utrecht University, The Netherlands
{k.kogkalidis,o.melkonian}@students.uu.nl

**Abstract.** We discuss the open problem of parsing the Dyck language of 3 symbols, $D^3$, using a 2-Multiple Context-Free Grammar. We attempt to tackle this problem by implementing a number of novel meta-grammatical techniques and present the associated software packages we developed.

**Keywords:** Dyck Language · Multiple context-free grammars (MCFG) · Young Tableaux · Spider Webs · Meta-grammars

## 1 Introduction

Multidimensional Dyck languages [8] generalize the well-known pattern of well-bracketed pairs of parentheses to k-symbol alphabets. Our goal in this paper is to study the 3-dimensional Dyck language $D^3$, and the question of whether this is a 2-dimensional multiple context-free language, 2-MCFL.

For brevity's sake, this section only serves as a brief introductory guide towards relevant papers, where the interested reader will find definitions, properties and various correspondences of the problem.

### 1.1 Preliminaries

We use $D^3$ to refer to the Dyck language over the lexicographically ordered alphabet $a < b < c$, which generalizes well-bracketed parentheses over three symbols. Denoting with $\#x(w)$ the number of occurrences of symbol $x$ within word $w$, a word belongs in $D^3$ if and only if it satisfies the following conditions:

(D1) $\#a(w) = \#b(w) = \#c(w)$
(D2) $\#a(v) \geq \#b(v) \geq \#c(v),\ \forall\ v \in \mathit{PrefixOf}(w)$

Eliding the second condition (D2), we get the $MIX$ language, which models free word order over the same alphabet. $MIX$ has already been proven expressible by a 2-MCFG [14]; the class of multiple context-free grammars that operate on pairs of strings [15].

### 1.2 Motivation

***Static Analysis*** Interestingly, the 2-symbol Dyck language ($D^2$) is used in the *static analysis* of programming languages, where a large number of analyses are formulated as *language-reachability* problems [12].

For instance, when considering interprocedural calls as part of the source language, high precision can only be achieved by examining only control-flow paths that respect the fact that a procedure call always returns to the site of its current caller [13]. By associating the program point *before* a procedure call $f_k$ with $(_k$, and the one *after* the call with $)_k$, the validity problem is reduced to recognizing $D^2$ words.

Alas, the 2-dimensional case cannot accommodate richer control-flow structures, such as exception handling via `try/catch` and Python generators via the `yield` keyword. To achieve this, one must lift the Dyck-reachability problem to a higher dimension which, given the computational cost that context-sensitive parsing induces, is currently prohibited. If $D^3$ is indeed a 2-MCFL, parsing it would become computationally attainable for these purposes and eventually allow scalable analysis for non-standard control-flow mechanisms by exploiting the specific structure of analysed programs, as has been recently done in the 2-dimensional case [1].

Last but not least, future research directions will open up in a multitude of analyses that are currently restrained to two dimensions, such as *program slicing*, *flow-insensitive points-to analysis* and *shape approximation* [12].

***Linguistics*** For the characterization of natural language grammars, the extreme degree of 'scrambling' permitted by the $MIX$ language may be considered overly expressive [4]. On the other hand, the prefix condition of $D^3$ allows for partial word movement, while still respecting certain linear order constraints, as observed in natural languages.

Supported by the fact that the language of well-bracketed parentheses, $D^2$, is a simple CFL (i.e. 1-MCFL) and given that $MIX$ itself is a 2-MCFL, it is reasonable to examine whether $D^3$ can also be modelled by a 2-MCFG. Such an endeavour proved quite challenging, necessitating careful study of correspondences with other mathematical constructs.

### 1.3 Correspondences

***Young Tableaux*** A standard Young Tableau is defined as an assortment of $n$ boxes into a ragged (or jagged, i.e. non-rectangular) matrix containing the integers 1 through $n$ and arranged in such a way that the entries are strictly increasing over the rows (from left to right) and columns (from top to bottom). Reading off the entries of the boxes, one may obtain the *Yamanouchi* word by placing (in order) each character's index to the row corresponding to its lexicographical ordering.

In the case of $D^3$, the Tableau associated with these words is in fact *rectangular* of size $n \times 3$, and the length of the corresponding word (called a *balanced*

*or dominant Yamanouchi word* in this context) is $3n$, where $n$ is the number of occurrences of each unique symbol [8]. Practically, the rectangular shape ensures constraint (D1), while the ascending order of elements over rows and columns ensures constraint (D2). In that sense, a rectangular standard Young tableau of size $n \times 3$ is, as a construct, an alternative way of uniquely representing the different words of $D^3$. We present an example tableau in Fig. 1.

| | | | | | | |
|---|---|---|---|---|---|---|
| a: | 1 | 3 | 4 | 8 | 9 | 10 |
| b: | 2 | 5 | 7 | 11 | 13 | 15 |
| c: | 6 | 12 | 14 | 16 | 17 | 18 |

**Fig. 1.** Young tableau for *"abaabcbaaabcbcbccc"*

***Promotions and Orbits*** There is an interesting transformation on Young Tableaux, namely the *Jeu-de-taquin* algorithm. When operating on a rectangular tableau $T(n, 3)$, Jeu-de-taquin consists of the following steps:

(1) Reduce all elements of T by 1 and replace the first item of the first row with an empty box $\square(x, y) := (1, 1)$.
(2) While the empty box is not at the bottom right corner of T, $\square(x, y) \neq (n, 3)$, do:
   - Pick the minimum of the elements directly to the right and below the empty box, and swap the empty box with it. $T(x, y) := min(T_{(x+1,y)}, T_{(x,y+1)})$, $\square(x', y') := (x + 1, y)$ (in the case of a right-swap) or $\square(x', y') := (x, y + 1)$ (in the case of a down-swap).
(3) Replace the empty box with $3n$.

The tableau obtained through Jeu-de-taquin on T is called its promotion $p(T)$. We denote by $p^k(T)$, $k$ successive applications of Jeu-de-taquin. It has been proven that $p^{3n}(T) = T$ [10]. In other words, the promotion defines an equivalence class, which we name an *orbit*, which cycles back to itself. Orbits dissect the space of $D^3$ into disjoint sets, i.e. every word $w$ belongs to a particular orbit, obtained by promotions of $T_w$.

**$\mathbf{A_2}$ *Combinatorial Spider Webs*** The $A_2$ *irreducible combinatorial spider web* is a directed planar graph embedded in a disk that satisfies certain conditions; we refer the reader to [5] for a formal definition. Spider webs can be obtained through the application of a set of rules, known as the *Growth Algorithm* [10]. These operate on pairs of neighbouring nodes, collapsing them into a singular intermediate node, transforming them into a new pair or eliminating them altogether. Growth rules will be examined from a grammatical perspective in Section 2.2. Upon reaching a fixpoint, the growth process produces a well-formed Spider Web, which, in the context of $D^3$, can be interpreted as a visual representation of parsing a word [8,10].

A bijection also links Young Tableaux with Spider Webs. More specifically, the act of promotion is isomorphic to a combinatorial `action` on spider webs, namely *web rotation* [10].

***Constrained Walk*** A Dyck word can also be visualized as a constrained *walk* within the first quadrant of $\mathbb{Z}^2$. We can assign each alphabet symbol $x$ a vector value $\boldsymbol{v_x} \in \mathbb{Z}^2$ such that all pairs of $(\boldsymbol{v_x}, \boldsymbol{v_y})$ are linearly independent and:

$$\boldsymbol{v_a} + \boldsymbol{v_b} + \boldsymbol{v_c} = \boldsymbol{0} \tag{1}$$

$$\kappa\boldsymbol{v_a} + \lambda\boldsymbol{v_b} + \mu\boldsymbol{v_c} \geq \boldsymbol{0}, (\forall\ \kappa \geq \lambda \geq \mu) \tag{2}$$

We can then picture Dyck words as routes starting from $(0, 0)$. (1) means that each route must also end at $(0, 0)$ ($\cong$ (D1)), while (2) means that the $x$ and $y$ axes may never be crossed ($\cong$ (D2)). An example walk is depicted in Fig. 2.

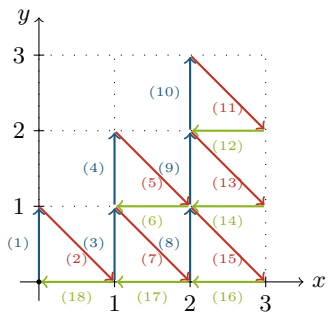


**Fig. 2.** The constrained walk of *"abaabcbaaabcbcbccc"* with vector value assignments $\boldsymbol{v}_a = (1, 0)$, $\boldsymbol{v}_b = (-1, 1)$, $\boldsymbol{v}_c = (0, -1)$

## 2 Abstract Grammar Specification

We now present a number of novel techniques that we developed as an attempt to solve the problem at hand, incrementally moving towards more complex and abstract grammars. For the purpose of experimentation we have implemented these techniques, based on a software library for parsing MCFGs [7]. The resulting Python code is open-source and available online[1].

### 2.1 Triple Insertion

To set things off, we start with the grammar of *triple insertion* in Fig. 3. This grammar operates on non-terminals $\mathrm{W}(x, y)$, producing $\mathrm{W}(x', y')$ with an additional triplet $a$, $b$, $c$ that respects the partial orders $x < y$ and $a < b < c$. The end-word is produced through the concatenation of $(x, y)$.

[1] https://github.com/omelkonian/dyck

$$\mathrm{S}(xy) \leftarrow \mathrm{W}(x, y). \tag{1}$$
$$\mathrm{W}(\epsilon, xy\mathbf{abc}) \leftarrow \mathrm{W}(x, y). \tag{2}$$
$$\mathrm{W}(\epsilon, x\mathbf{a}y\mathbf{bc}) \leftarrow \mathrm{W}(x, y). \tag{3}$$
$$\mathrm{W}(\epsilon, x\mathbf{ab}y\mathbf{c}) \leftarrow \mathrm{W}(x, y). \tag{4}$$
$$\dots$$
$$\mathrm{W}(\mathbf{abc}xy, \epsilon) \leftarrow \mathrm{W}(x, y). \tag{61}$$
$$\mathrm{W}(\epsilon, \mathbf{abc}). \tag{62}$$
$$\mathrm{W}(\mathbf{a}, \mathbf{bc}). \tag{63}$$
$$\mathrm{W}(\mathbf{ab}, \mathbf{c}). \tag{64}$$
$$\mathrm{W}(\mathbf{abc}, \epsilon). \tag{65}$$

**Fig. 3.** Grammar of triple insertions

Despite being conceptually simple, this grammar consists of a large number of rules. Its expressivity is also limited; the prominent weak point is its inability to manage the effect of *straddling*, namely the generation of words whose substituents display complex interleaving patterns. Refer to Fig. 10 for an example.

### 2.2 Meta-Grammars

To address the issue of rule size, we employ the notion of *meta-grammars*, loosely inspired by Van Wijngaarden's work [16], which allows a more abstract view of the grammar as a whole. Meta-grammars have found wide use in describing linguistic phenomena involving discontinuity and word permutation, giving rise to abstraction techniques that alleviate the need to make all orderings explicit, such as partially ordered multi-set CFGs [9], phrase structure grammars [2,3] and the domain union operation [11].

Specifically, we define $\mathcal{O}$ as the *meta-rule* which, given a rule format, a set of partial orders (over the tuple indices of its premises and/or newly added terminal symbols), and the MCFG dimensionality, automatically generates all the order-respecting permutations. An example of how we can abstract away from explicitly enumerating the entirety of our initial rules is showcased in Fig. 4.

$$\mathrm{S}(xy) \leftarrow \mathrm{W}(x, y).$$
$$\mathcal{O}_2[\![\mathrm{W} \leftarrow \epsilon \mid \{a < b < c\}]\!].$$
$$\mathcal{O}_2[\![\mathrm{W} \leftarrow \mathrm{W} \mid \{x < y,\ a < b < c\}]\!].$$

**Fig. 4.** $\mathcal{G}_0$: Meta-grammar of triple insertions

This approach enhances the potential expressivity of our grammars as well. For instance, we can now extend the previous grammar with a single meta-rule that allows two non-terminals $\mathrm{W}(x, y)$, $\mathrm{W}(z, w)$ to interleave with one another, producing rearranged tuple concatenations and allowing some degree of straddling to be generated:

$$\mathcal{G}_1 : \mathcal{G}_0 + \mathcal{O}_2[\![\mathrm{W} \leftarrow \mathrm{W}, \mathrm{W} \mid \{x < y,\ z < w\}]\!].$$

The addition of this rule gets us closer to completeness, but we are still not quite there. We have thus far only used a single non-terminal, not utilizing the expressivity that an MCFG allows. To that end, we propose new non-terminals to represent incomplete word *states*; that is, words that either have an extra symbol or miss one. The former are *positive* states, whereas the latter are *negative*. The inclusion of these extra states would allow for more intricate interactions, in line with the CFG that describes the 2-letter $MIX$ language [6].

Interestingly, there is a direct correspondence between these non-terminals and the nodes of Petersen's growth algorithm [10]. Fig. 5 depicts the growth rules in the exact same web form as proposed by Petersen, modulo node branding. Briefly, these describe a way of constructing minimal parses of $D^3$ words, as follows:

1. each character in the original string gets translated into a positive state
2. each growth rule combines adjacent elements on the top, either producing one/two new outputs on the bottom or cancelling them out
3. the growth process terminates (i.e. the word is parsed) when there remain no dangling strands

A subset of these web-reduction rules are, in fact, precisely modelled by the meta-grammar $\mathcal{G}_2$ presented in Fig. 6. In section 4, we briefly explain our inability to model the whole set of rules with a 2-MCFG, which would render our grammar complete.

$\mathcal{G}_2$ consists of base cases for positive states, possible state interactions, closures of pairs of inverse polarity and a universally quantified meta-rule that allows the combination of any incomplete state with a well-formed one (i.e. non-terminal W).

A further extension can be achieved through universally quantifying the notion of triple insertion, which is unique in the sense that it can insert three different terminals, each at a different position:

$$\mathcal{G}_3 : \mathcal{G}_2 + \forall\ \mathrm{K} \in \{\mathrm{A}^{+/-},\ \mathrm{B}^{+/-},\ \mathrm{C}^{+/-}\} : \mathcal{O}_2[\![\mathrm{K} \leftarrow \mathrm{K} \mid \{x < y,\ a < b < c\}]\!].$$

### 2.3 Rule Inference

The improved performance of the above approaches again proved insufficient to completely parse $D^3$. Our meta-rules are over-constrained by imposing a total order on the tuple elements, due to their inability to keep track of where the extra character(s) is. To overcome this, we split each state into multiple position-aware, *refined* states. Doing so revealed a vast amount of new interactions, as

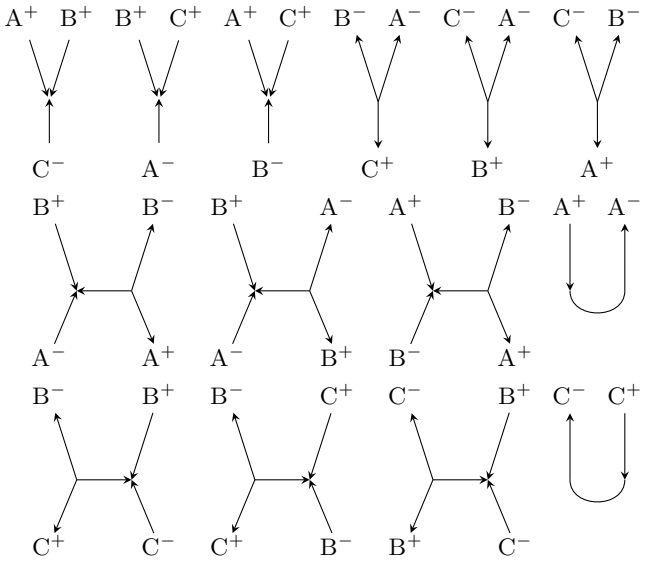


**Fig. 5.** Growth rules

evidenced by the below alteration to the original $A^+$, $B^+$ interaction (where $y$ can now occur after $z$ or $w$):

$$\mathcal{O}_2[\![C^- \leftarrow A^+_{left}, B^+ \mid \{x < y, x < z < w\}]\!].$$

In order to accommodate the interactions between this increased number of states, we need to keep track of both internal and external order constraints. At this point, the abstraction offered by our meta-grammar approach does not cover our needs anymore. The same difficulty that we had encountered before is prominent once more, except now at an even higher level.

As a solution to the aforementioned limitation, we propose a system that can automatically create a full-blown m-MCFG given only the states it consists of. To accomplish this, we assign each state a unique *descriptor* that specifies the content of its tuple's elements. Aligning these descriptors with the tuple, we can then infer the descriptor of the resulting tuple of every possible state interaction. For the subset of those interactions whose resulting descriptor is matched with a state, we can now automatically infer the rule.

Formally, the system is initialized with a map $\mathcal{D}$, such as the one illustrated in Fig. 7. Its domain, $dom(\mathcal{D})$, is a set of *state identifiers* and its codomain, $codom(\mathcal{D})$, is the set of their corresponding *state descriptors*.

Meta-grammars accelerated the process of creating grammars, by letting us simply describe rules instead of explicitly defining them. ARIS builds upon this notion to raise the level of abstraction even further[2]; one needs only specify a

[2] An avid reader will notice the use of the $\mathcal{O}$ meta-rule in the ARIS algorithm.

$$
\begin{aligned}
&\mathrm{S}(xy) \leftarrow \mathrm{W}(x,y).\\
&\mathcal{O}_2[\![\mathrm{W} \leftarrow \epsilon \mid \{a<b<c\}]\!].\\
&\mathcal{O}_2[\![\mathrm{A}^+ \leftarrow \epsilon \mid \{a\}]\!].\\
&\mathcal{O}_2[\![\mathrm{B}^+ \leftarrow \epsilon \mid \{b\}]\!].\\
&\mathcal{O}_2[\![\mathrm{C}^+ \leftarrow \epsilon \mid \{c\}]\!].\\
&\mathcal{O}_2[\![C^- \leftarrow A^+, B^+ \mid \{x<y<z<w\}]\!].\\
&\mathcal{O}_2[\![B^- \leftarrow A^+, C^+ \mid \{x<y<z<w\}]\!].\\
&\mathcal{O}_2[\![A^- \leftarrow B^+, C^+ \mid \{x<y<z<w\}]\!].\\
&\mathcal{O}_2[\![A^+ \leftarrow C^-, B^- \mid \{x<y<z<w\}]\!].\\
&\mathcal{O}_2[\![B^+ \leftarrow C^-, A^- \mid \{x<y<z<w\}]\!].\\
&\mathcal{O}_2[\![C^+ \leftarrow B^-, A^- \mid \{x<y<z<w\}]\!].\\
&\mathcal{O}_2[\![W \leftarrow A^+, A^- \mid \{x<y<z<w\}]\!].\\
&\mathcal{O}_2[\![W \leftarrow C^-, C^+ \mid \{x<y<z<w\}]\!].\\
&\forall\ \mathrm{K} \in \{\mathrm{A}^{+/-},\ \mathrm{B}^{+/-},\ \mathrm{C}^{+/-}\}:\\
&\quad \mathcal{O}_2[\![\mathrm{K} \leftarrow \mathrm{K}, \mathrm{W} \mid \{x<y,\ z<w\}]\!].
\end{aligned}
$$

**Fig. 6.** $\mathcal{G}_2$: Meta-grammar of incomplete states

$$
\begin{aligned}
\mathrm{W} &\mapsto (\epsilon,\epsilon) & \mathrm{A}_r^- &\mapsto (\epsilon, bc)\\
\mathrm{A}_l^+ &\mapsto (a,\epsilon) & \mathrm{A}_{lr}^- &\mapsto (b,c)\\
\mathrm{A}_r^+ &\mapsto (\epsilon,a) & \mathrm{B}_l^- &\mapsto (ac,\epsilon)\\
\mathrm{B}_l^+ &\mapsto (b,\epsilon) & \mathrm{B}_r^- &\mapsto (\epsilon,ac)\\
\mathrm{B}_r^+ &\mapsto (\epsilon,b) & \mathrm{B}_{lr}^- &\mapsto (a,c)\\
\mathrm{C}_l^+ &\mapsto (c,\epsilon) & \mathrm{C}_l^- &\mapsto (ab,\epsilon)\\
\mathrm{C}_r^+ &\mapsto (\epsilon,c) & \mathrm{C}_r^- &\mapsto (\epsilon,ab)\\
\mathrm{A}_l^- &\mapsto (bc,\epsilon) & \mathrm{C}_{lr}^- &\mapsto (a,b)
\end{aligned}
$$

**Fig. 7.** Map $\mathcal{D}$ for refined states

**Algorithm 1** ARIS: Automatic Rule Inference System

**procedure** ARIS($\mathcal{D}$)
  **for** $X \mapsto (d_1, \ldots, d_n) \in \mathcal{D}$ **do**
    **yield** $X(d_1, \ldots, d_n)$.
  **for** $X, Y \in dom(\mathcal{D})^2$ **do**
    $(X_{ord},\ Y_{ord}) \leftarrow (x < y < \ldots,\ z < w < \ldots)$
    **for** $(d_1, ..., d_n) \in \mathcal{O}_2[\![\_ \leftarrow X, Y \mid \{X_{ord}, Y_{ord}\}]\!]$ **do**
      **for** $S' \in$ ELIMINATE$((d_1, \ldots, d_n), \mathcal{D})$ **do**
        **yield** $S'(d_1, \ldots, d_n) \leftarrow X, Y$.

**procedure** ELIMINATE$((d_1, \ldots, d_n), \mathcal{D})$
  **for** $matches \in$ ALL_ABC_TRIPLETS$(d_1, \ldots, d_n)$ **do**
    **for** $i \in 0 \ldots n/3$ **do**
      **for** $S' \in$ REMOVE_ABC_TRIPLETS$(matches, i)$ **do**
        **if** $S' \in codom(\mathcal{D})$ **then**
          **yield** $S'$

grammar's states and its descriptors, thus eliminating the need to define rules or even meta-rules.

We use ARIS instantiated with map $\mathcal{D}$ of Fig. 7 to generate $\mathcal{G}_4$ as our last attempt to parse $D^3$. Even though map $\mathcal{D}$ consists of a mere amount of 16 mappings, it produces a lavish parsing system of 1456 concrete rules; disappointingly, these again do not yield a complete solution to our problem.

## 3 Tools & Results

### 3.1 Grammar Utilities

We have implemented the modelling techniques described in Section 2 and distributed a Python package, called **dyck**, which provides the programmer with a *domain-specific language* close to this paper's mathematical notation. To facilitate experimentation, our package includes features such as grammar selection, time measurements, word generation and soundness/completeness checking. The following example demonstrates the definition of $\mathcal{G}_3$:

```
from dyck import *
G3 = Grammar(initial='W',
  # Base Cases
  O('W', {(a, b, c)}),
  O('A-', {(b, c)}), O('B-', {(a, c)}), O('C-', {(a, b)}),
  O('A+', {(a,)}),   O('B+', {(b,)}),   O('C+', {(c,)}),
  # Combinations
  O('C- ← A+, B+', {(x, y, z, w)}),
  O('B- ← A+, C+', {(x, y, z, w)}),
```

```
O('A- ← B+, C+', {(x, y, z, w)}),
O('C+ ← B-, A-', {(x, y, z, w)}),
O('B+ ← C-, A-', {(x, y, z, w)}),
O('A+ ← C-, B-', {(x, y, z, w)}),
forall(all_states,
       lambda K: O('K ← K, W', {(x, y), (z, w)})),
# Closures
O('W ← A+, A-', {(x, y, z, w)}),
O('W ← C-, C+', {(x, y, z, w)}),
# Universal Triple Insertion
forall(all_states,
       lambda K: O('K ← K', {(x, y), (a, b, c)})))
```

### 3.2 Visualization

As counter-examples began to grow in size and number, we realised the necessity of a visualization tool to assist us in identifying properties they may exhibit. To that end, we distribute another Python package, called **dyckviz**, which allows the simultaneous visualization of tableau-promotion and web-rotation (grouped in their corresponding equivalence classes). An example of a web as rendered by our tool is given in Fig. 8.

Young tableaux in an orbit are colour-grouped by their column indices, which sheds some light on how the *jeu-de-taquin* actually influences the structure of the corresponding Dyck words. Interesting patterns have began to emerge, which still remain to be properly investigated.

### 3.3 Grammar Comparisons

Fig. 9 displays three charts, depicting the number of rules, percentage of counter-examples and computation times of each of our grammars for $D_n^3$ with $n$ ranging from 2 to 6 (where $n$ denotes the number of *abc* triplets). Even though none of our proposed grammars is complete, we observe that as grammars get more abstract, the number of failing parses steadily declines. This however comes at the cost of rule size growth, which in turn is associated with an increase in computation times. What this practically means is that we are unable to continue testing more elaborate grammars or scale our results to higher orders of $n$ (note that $\|D_n^3\|$ also has a very rapid rate of expansion[3]).

## 4 Discussion

To our knowledge, no other attempt has come as close to modelling $D^3$ with a 2-MCFG. We attribute this to the combination of a pragmatic approach with results from existing theoretical work. In this section, we present a collection of additional ideas, which we consider worthy of further exploration.

[3] https://oeis.org/A000108

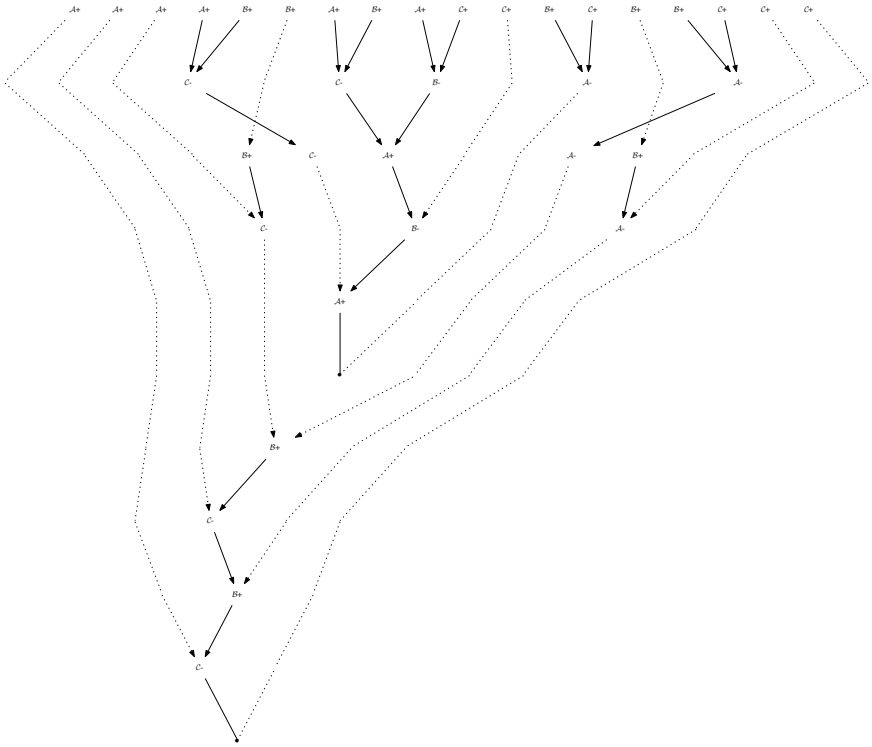

**Fig. 8.** Spider web of *"abaacbbacbabaccbcc"*

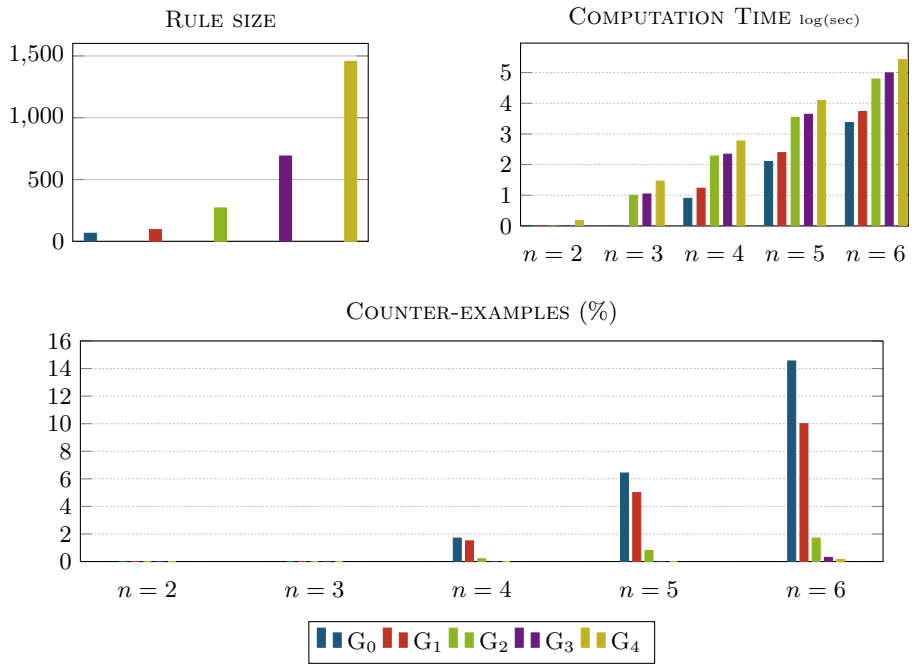


**Fig. 9.** Performance measures

***First-Match Policy and Relinking*** Possibly the most intuitive way of checking whether a word $w$ is part of $D_n^3$ is checking whether a pair of links occur that match $a_i$ to $b_i$ and $b_i$ to $c_i$, $\forall\ i \in n$. We call this process of matching the *first-match policy*. The question arises whether a grammar can accomplish inserting a triplet of $a$, $b$, $c$, that would abide by the first-match policy. If that were the case, it would be relatively easy to generalize this ability by induction to every $n \in \mathbb{N}$. Unfortunately, the answer is seemingly negative; the expressiveness provided by a 2-MCFG does not allow for the arbitrary insertions required. On a related note, being able to produce a word state $W(x, y)$ where $w = xy$ and $x$ any possible prefix of $w$, gives no guarantee of being able to produce the same word with an extra triplet inserted due to the straddling property.

However, if rules existed that would allow for match-making and breaking, i.e. match *relinking*, an inserted symbol could be temporarily matched with what might be its first match-policy in a local scope, and then relink it to its correct match when merging two words together.

***Growth Rules*** Although $\mathcal{G}_2$ comes close to realizing the growth algorithm, not all of the growth rules presented in Figure 5 can be translated into a 2-MCFG setting. In particular, there is no straight-forward way to render the growth rules which produce two edges (i.e. the six bottom-left rules); two neighbouring edges


**Fig. 10.** First-match policy for *"ababacbcabcc"*

can in turn interact with their next neighbours in a recursive fashion, which can not be accommodated by our current tuple representation. We cannot, however, rule out the possibility of the growth rules being scalably translatable into m-MCFG rules without resorting to more expressive formalisms (e.g. context-sensitive rewriting systems). In fact, such a translation would be a guarantee of completeness.

***Insights from promotion*** An interesting question is whether promotion can be handled by a 2-MCFG. If so, it could be worth looking into the properties of orbits, to test for instance if there are promotions within an orbit that can be easier to solve than others. Solving a single promotion and transducing the solution to all equivalent words could then be a guideline towards completeness.

## 5 Conclusion

We tried to accurately present the intricacies of $D^3$ and the difficulties that arise when attempting to model it as a 2-MCFL. We have developed and introduced some novel techniques and tools, which we believe can be of use even outside the problem's narrow domain. We have largely expanded on the existing tools to accommodate MIX-style languages and systems of meta-grammars in general.

Despite our best efforts, the question of whether $D^3$ can actually be encapsulated within a 2-MCFG still remains unanswered. Regardless, this problem has been very rewarding to pursue, and we hope to have intrigued the interested reader enough to further research the subject, use our code, or strive for a solution on her own.

## Acknowledgements

We would like to thank Michael Moortgat for introducing us to the problem, providing insightful feedback and motivating us throughout the process, as well as Jurriaan Hage for suggesting the use of multi-dimensional Dyck languages in static program analysis.

## References

1. Chatterjee, K., Choudhary, B., Pavlogiannis, A.: Optimal Dyck reachability for data-dependence and alias analysis. Proceedings of the ACM on Programming Languages 2(POPL), 30 (2017)

2. Gazdar, G.: Phrase structure grammar. In: The nature of syntactic representation, pp. 131–186. Springer (1982)
3. Gazdar, G., Klein, E., Pullum, G.K., Sag, I.A.: Generalized phrase structure grammar. Harvard University Press (1985)
4. Kanazawa, M., Salvati, S.: MIX is not a tree-adjoining language. In: Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1. pp. 666–674. Association for Computational Linguistics (2012)
5. Kuperberg, G.: Spiders for rank 2 Lie algebras. Communications in mathematical physics 180(1), 109–151 (1996)
6. Lewis, H.R., Papadimitriou, C.H.: Elements of the Theory of Computation. Prentice Hall PTR (1997)
7. Ljunglöf, P.: Practical parsing of parallel multiple context-free grammars. In: Workshop on Tree Adjoining Grammars and Related Formalisms. p. 144 (2012)
8. Moortgat, M.: A note on multidimensional Dyck languages. In: Categories and Types in Logic, Language, and Physics. pp. 279–296 (2014)
9. Nederhof, M.J., Shieber, S., Satta, G.: Partially ordered multiset context-free grammars and ID/LP parsing. Association for Computational Linguistics (2003)
10. Petersen, T.K., Pylyavskyy, P., Rhoades, B.: Promotion and cyclic sieving via webs. Journal of Algebraic Combinatorics 30(1), 19–41 (2009)
11. Reape, M.: Getting things in order. Discontinuous constituency 6, 209–253 (1996)
12. Reps, T.: Program analysis via graph reachability. Information and software technology 40(11-12), 701–726 (1998)
13. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 49–61. ACM (1995)
14. Salvati, S.: MIX is a 2-MCFL and the word problem in $Z^2$ is solved by a third-order collapsible pushdown automaton. Journal of Computer and System Sciences 81(7), 1252–1277 (2015)
15. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science 88(2), 191–229 (1991)
16. van Wijngaarden, A.: The generative power of two-level grammars. In: ICALP (1974)