D^3 as a 2-MCFL

Orestis Melkonian, Konstantinos Kogkalidis
Utrecht University, The Netherlands
o.melkonian@uu.nl, konstantinos@riseup.net

Abstract

We discuss the problem of parsing the Dyck language of 3 symbols, D^3, using a 2-Multiple Context-Free Grammar. We implement a number of novel techniques designed for the purpose of solving that problem and present the associated software package we developed.

CCS Concepts • **Formal languages and automata theory** → **Grammars and context-free languages**

Keywords Dyck Language; Multiple Context Free Grammars (MCFG); Parsing

> When you depart for Ithaca,
> wish for the road to be long,
> full of adventure, full of knowledge.
>
> *C. P. Cavafy*
> *Ithaca*

1. Introduction

Our goal with this paper is the analysis of the 3-dimensional Dyck Language, D^3 under the scope of a 2-multiple context-free language, 2-MCFL. In this chapter, we start by providing some concise definitions and a brief overview of interesting correspondences and properties of the D^3. The goal of this chapter is to simply introduce the reader to some possibly less known concepts rather than to fully extend upon them. For a deeper and more detailed view of the ideas presented, we direct the interested reader towards our list of references, which provide significantly more information.

1.1 Definitions

MIX Borrowing notation from Kazanawa, we define the MIX_d language as the language over a d-symbol, lexigraphically ordered alphabet $a_1 < a_2 < ...a_d$, whose words w satisfy the condition:

(D1) $\#a_i(w) = \#a_j(w) \; \forall \; i, j \in d$

where $\#x(w)$ refers to the number of occurrences of symbol x within word w.

Dyck Language We use D^d to refer to the Dyck language over the same alphabet. D^d is defined as a subset of MIX_d, whose words w abide to one additional constraint:

(D2) For every prefix v of w, $\#a_i(v) \geq \#a_j(v) \; \forall i, j \; \in d$ such that $i < j$.

The D^2 language corresponds to the language of well-bracketed parentheses, and D^d generalizes this over an arbitrarily large alphabet.

MCFL Multiple context free languages, abbreviated MCFLs, is the class of languages generated by multiple context free grammars, abbreviated MCFGs. Multiple context-free grammars are a mildly context-sensitive grammar formalism that, rather than strings, operate on tuples of strings. MCFGs contain partial functions defined as the concatenation of constant strings from the terminal alphabet and components of their arguments Gotzmann. An m-multiple context-free grammar, m-MCFG, acts upon tuples of at most m-elements. The language generated by an m-MCFG is an m-MCFL. The class of MCFLs is interesting to the domain of linguistics, because it is believed to capture the intricacies of natural languages.

1.2 Correspondences

Young Tableaux A standard Young Tableau is defined as an assortment of n boxes into a ragged (or jagged, ie. 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 $w_Y = w_1 w_2 ... w_n$ with w_i corresponding to symbol a_j, where j is the row containing entry i.

In the case of D^d, the Tableau associated with these words is in fact *rectangular* of size $n \times d$, and the length of the corresponding word (called a *balanced or dominant Yamanouchi word* in this context) is dn, where n is the number of occurrences of each unique symbol and d the total number of unique symbols Moortgat. 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 tableu of size $n \times 3$ is, as a construct, an alternative way of uniquely representing the different words of D^3. We present two examples below:

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

Table 1. Young tableau for *abaabcbaaabcbcbccc*

Promotions and Orbits Having presented the concept of a Young Tableau, we can now move ahead with the *Jeu-de-taquin* algorithm. When operating on a rectangular Table $T(n, d)$, 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, $\square(x, y) \neq (n, d)$, do:

  (a) 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 dn.

Table 2. Jeu-de-taquin algorithm for rectangular Tableux

The table obtained through Jeu-de-taquin on T is called its promotion $p(T)$. k successive applications of Jeu-de-taquin result in $p^k(T)$. M. Haiman shows that that $p^{dn}(T) = T$. 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_n^d into disjoint sets, ie. there is no word w whose Tableu T_w belongs to more or less than a single orbit.

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 $\vec{v_x} \in \mathbb{Z}^2$ such that all pairs of $(\vec{v_x}, \vec{v_y})$ are linearly independent and:

$$\vec{v_a} + \vec{v_b} + \vec{v_c} = \vec{0} \quad (1)$$

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

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 follows:

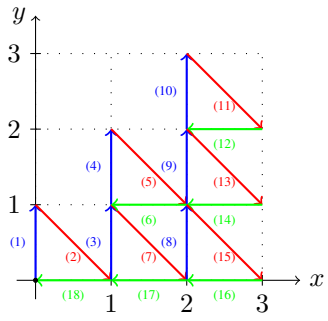

Figure 1. The constrained walk of *abaabcbaaabcbcbccc* with vector value assignments $\vec{v_a} = (1, 0)$, $\vec{v_b} = (-1, 1)$, $\vec{v_c} = (0, -1)$

A_2 ***Combinatorial Spider Webs*** The A_2 *irreducible combinatorial spider web* is a category which Khu defines as a planar directed graph D, with no multiple edges embedded in a disk that satisfies conditions W1 through W3:

(W1) D consists of positive and negative edges, with each edge being oriented from one of the negative edges to one of the positive edges.

(W2) All boundary vertices have degree 1 while all internal vertices have degree 3.

(W3) D is non-elliptic, ie. all internal faces of D have at least 6 faces.

A_2 is a way of constructing visually intriguing, minimal parses of D^3 words. To make this clear, we first define the three *positive* and three *negative* edges which assume the role of states in the parsing process of D^3, aligned with their corresponding alphabet symbols (or sequence thereof):

State	Symbol Sequence
A^+	a
B^+	b
C^+	c
A^-	bc
B^-	ac
C^-	bc

Table 3. Edges of the A_2 growth algorithm

We now present a version of Petersen's summation of the *growth algorithm* in web form, which models the various possible interactions of edges-states, rephrased to our problem's scope.

Growth Rules

The edges on the upper level can be seen as input nodes which, interacting with one another, produce either one output edge or a pair thereof. The soundness of each individual *growth rule* is immediately obvious. On a higher level, what the growth algorithm achieves is to:

(a) Merge two neighbouring edges of the same polarity into one of the opposite, and propagate it downwards.

(b) Perform the closures of $\{A+, A-\}$ and $\{C-, C+\}$.

(c) Swap the horizontal position of pairs of states between which no immediate interaction exists, such that either of

them eventually ends up neighbouring another edge with which it can be eliminated.[1]

At termination, the growth algorithm produces an A_2 web with no *dangling strands* and no edge crossing Petersen.

1.3 Properties

Cardinality The number of words produced by D_n^3, $\|D_n^3\|$, corresponds to the A005789 integer sequence[2] [Moortgat]. Interestingly, the number of orbits $\|O(D)\|$ exhibits symmetricity between the alphabet size d and the number of occurrences n, ie. $\|O(D_d^n)\| = \|O(D_n^d)\|$.

$d\backslash n$	2	3	4	5	6
2	2	5	14	42	132
	1	2	3	6	14
3	5	42	462	6006	87516
	2	6	44	406	4896

Table 4. Word and orbit (grayscale) cardinality of D_n^d

Non-wellnestedness The property of well-nestedness can be informally described as the absence of rules that permute the tuples of interacting states in such a way that their corresponding dependency trees interleave Joshi 1997. A more formal definition is given by Salvati, MIX, which we will not include. In the case of D^3, there seem to exist cross-serial dependencies, violating the above constraint, and rendering D^3 non-wellnested.

2. Modeling Techniques

Having established the necessary vocabulary, we can now begin tackling the elusive problem of parsing D^3 via a 2-MCFG. This chapter's contents follow a chronological order, tracking our progressively larger, more complex and more abstract grammars.

2.1 Triple Insertion

During our early days of initiation to the world of D^3, the problem seemed only as a natural continuation to the (much simpler) ones we had encountered before. With its difficulty not immediately apparent, our first modeling attempt was rather naive. Yet despite its naivety, it is still a relevant component of the grammars to follow, so we present it here.

The grammar of *triple insertion*, as its name suggests, simply inserts in order the three alphabet symbols *a, b* and *c* in any way possible given a non-terminal state W (for *word*) consisting of the tuple (x, y). This non-terminal produces a variety of tuples that respect both (D1) and (D2). The end-word is produced through the concatenation of (x, y) as produced by a single-arity state $S(xy)$.

[1] This property is referred to as crossing removal. Petersen shows that the outcome of crossing removal is uniquely determined and independent of the order of removal.

[2] Refer to the Online Encyclopedia of Integer Sequences (*OEIS*)

Grammar
Counter-Example

Despite its compact size (and inarguable naivety), our first grammar is nevertheless relatively expressive. The prominent weak point is its inability to manage the effect of *straddling*, namely the generation of words that exhibit this characteristic...

2.2 Adding States: Meta-Grammars

The underwhelming performance of our first attempt signaled for the necessity of more expressive states that would allow for more complex, "longer range" interactions. Intuitively, we imagined a convention by which certain new states would exist. Through controlled interactions, pairs of these new, "incomplete" states could allow parts of words to propagate through the rules until they eventually met their closures or were transformed into a single, new state. In that spirit, we defined the six states of cross-reference table. One key difference is that our states may describe strings of arbitrary length, with the polarity now signifying whether the state is carrying an extra character or missing one instead (or having two extra characters, equivallently).

In hindsight, we were able to identify a differently represented version of our idea in the growth rules, reaffirming our initial idea that these new states were indeed sound and necessary. The added expressivity, however, came at a cost of rapid rule expansion. Each of the new states could now (potentially) interact with every other, and their means of interaction could become convoluted and hard to decipher or write down. To overcome this obstacle we needed to be able to describe our rules at a slightly more abstract level. For instance, a state $A^+(x, y)$ (which now models not just the terminal a but rather an intermediate state for which $\#b(xy) = \#c(xy) = \#a(xy) - 1$), could interact with a $B^+(z, w)$ to produce a C^- in any of the following ways. ways

Note that for each pair interaction, different constraints pertaining to the order of the permuted concatenated strings need to be preserved, to account for (D2). In the example above, for instance, the internal order of each state's tuple is crucial, but also the external order enforced by the particular pair (ie. the extra a must come before the extra b). For that purpose 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 the MCFG dimensionality, automatically generates all the allowable permutations. We thereby reduce the manual labour required to transcribe and debug individual rules.

$$\mathcal{O}[\![\mathrm{C}^- \leftarrow \mathrm{A}^+, \mathrm{B}^+ \mid \{x < y < z < w\}]\!]^{(2)}$$

We introduce the concise mathematical notation depicted above, which demonstrates that a state C^+ can be produced by two states $\mathrm{A}^+(x, y)$, $\mathrm{B}^+(z, w)$ by any 2-element tuple concatenation that respects the order (x,y,z,w), ie. the tuple

of C^- may any of $(\epsilon, xyzw)$, (x, yzw), (xy, zw), (xyz, w) or $(xyzw, \epsilon)$[3].

This additional abstraction, however, is not enough on its own. We still needed the ability to enforce additional constraints related not to the order but rather the positions of concatenated strings. To make this clear, let us present the unique case of B^-. Unlike other negative states, $B^-(x, y)$ has no direct closure unless we actually know it to contain its extra a in the left-side string x and its extra c in y (we refer to this distinction between a position-blind and a position-aware state as *refinement*). In that case alone can we insert the missing b (obtained from a $B^+(z, w)$ between x and y. But in order to automatically generate B^- states for which this additional condition holds, we need to keep note of the origin of the extra a and c, and filter out the rules generated by $\mathcal{O}$ accordingly. Similarly, we define $\mathcal{C}$ as a stricter version of $\mathcal{O}$, which constraints the rules according to the positions the premise tuples may occupy (after concatenation) in the conclusion's tuple.

$$\mathcal{C}[\![\mathrm{B}^- \leftarrow \mathrm{A}^+, \mathrm{C}^+ \mid \{x_l < y_l, z_r < w_r\}]\!]^{(2)}$$

In the above notation, we illustrate that B^- may be constructed from $A^+(x, y)$ and $C^+(z, w)$ only as long as the partial orders (x, y) and (z, w) are independently satisfied and x, y are both placed on the left element of the new state's tuples, while z and w on the right. This of course only allows for C^- to consist of the tuple (xy, zw). declarative programming

The above abstractions significantly accelarated the process of creating and testing grammars, as they allowed us to simply describe rules instead of explicitly defining them. We thus moved from the lower level of grammars to the higher level of *meta-grammars*, or grammars of grammars.

2.3 Refining States: Rule Inference

Perhaps expectedly, the improved performance of our new approach again proved insufficient to completely parse D^3. New problems started arising when trying to solve longer words; even though they were sound, our rules failed to make full use of the expressivity that a 2-MCFG allows. We soon realized that we have over-constrained our rules by making them comply to the worst-case scenario. Following the example of 2.2?, if we had kept memory of the position of a in $A^+(x, y)$, we could allow new rules to exist, such as rule in the case of a in x. Splitting each state into multiple different, *refined* states that now provide knowledge of their extra characters' positions, we could model vastly more interactions.

At this point, the abstraction offered by our meta-grammars did not cover our needs anymore. The same difficulty that we had encountered before was prominent once more, except now at an even-higher level. Transcribing meta-rules and proofreading through them was very time consuming and draining, and hindered our progress. In order to test a new hypothesis pertaining to the refined states we would again have to go through a lot of arid work.

As the ultimate solution for flexible grammar creation, we designed a system that can automatically create a full-blown 2-MCFG given only the states it consists of. To accomplish this, we assign each state a unique *descriptor* that describes (up to an extend) 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 thse interactions whose resulting descriptor is matched with a state, we can now automatically infer the rule. Algorithm

example

3. Implementation

Enabled by the machinery described in the previous chapter we performed a multitude of tests using different grammars. In this chapter we describe a few of these grammars that we deem as milestones of our progression, in ascending order of complexity. We also present some statistics to allow for a direct comparison to be made between them. For the purposes of this project we use a version of MCFParser [Ljunglf] as our backend, which we adapted to our needs[4].

3.1 Grammars

G_0: Triple Insertion This very simple grammar consists of simply the rules of triple-insertion on a single non-terminal state $W(x, y)$, as described in chapter 2.

grammar in all o notation

G_1: Adding word composition Benefiting from the added expressivity of all o, we can extend the previous grammar with a meta-rule that allows two non-terminals $W(x, y)$, $W(z, w)$ to interact with another, producing rearranged tuple concatenations and allowing some degree of straddling to be generated.

extra rule in all o

G_2: Incomplete words This grammar largely expands upon the concept of state interaction of the previous grammar, replacing the idea of triple insertion with the use of non-terminals that correspond to types of incomplete words. It was only after we had written it that we realized that the rules of this grammar are actually equivalent to the non-swapping rules of the growth algorithm.

grammar

G_3: Re-adding triple insertion Surprisingly, cases manageable by triple insertion could not be solved by the last grammar despite its elegance. After some thought, we realized that triple insertion is unique in the sense that it can insert three different terminals each at a different position,

[3] By ϵ we denote the empty string

[4] The original can be found at https://github.com/heatherleaf/MCFParser.py

something that no other pair interaction can directly accomplish. We thus allowed triple insertion back into our last grammar, now at the level of each new state.

grammar

G_4: Generalizing triple insertion The concept of triple insertion can be generalized; rather than simply adding terminals a, b and c, we could add larger word segments through states A^+, B^+ and C^+. To do so, we need to model the concurrent interaction of these three states with any other state. Our abstractions are actually able to manage that, so we test another grammar which replaces the triple insertion with a *triple state insertion*.

grammar

G_5: Refined non-terminals As the peak of our efforts, we replace non-terminals with their refined versions as presented in reference 2.3. $A^+(x, y)$ is now split into $A^+_{left}(x, y)$ and $A^+_{right}(x, y)$, with the extra a residing in x and y in each case respectively.

3.2 Results & Comparisons

Aspernatur fugit autem numquam corporis repellendus quasi. Et nisi veritatis consequatur adipisci iure. Sint quos aspernatur ea iste est.

Nobis temporibus nihil expedita quia blanditiis. Non voluptatum vel autem voluptatem occaecati libero. Aperiam nostrum id sed voluptatibus fuga. Culpa commodi molestiae molestiae corrupti nam. Sapiente quo commodi architecto exercitationem id ad. Non optio eos perferendis.

4. Road to completeness

As the above results suggest, despite our best efforts the problem is yet unsolved. In this chapter we present a collection of unexplored ideas which we consider noteworthy. These ideas are related to the concept of formally concluding the completeness of a grammar, bypassing the computational problem of parsing through words in search of an example the tested grammar fails to generate.

4.1 First-match policy and Relinking

Possibly the most intuitive way of checking whether a word w is part of D^3_n is checking whether a pair of links occur that match a_i to b_i and b_i to c_i $\forall i \in n$, where x_i the i-th occurrence of symbol x in w. We call this process of matching the *first-match policy*.

example

The question arises whether a grammar can accomplish inserting a triplet of a, b, c in such a way that the triplet inserted always corresponds to a triplet matched 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 rather 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 (ie. all different comma positions i for $i \in 0, 1...3n$, gives no guarantee of being able to produce the same word with an extra triplet inserted due to the straddling property. examples

However, if rules existed that would allow for match-making and breaking, ie. 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. This might be an interesting prospect to look into.

4.2 Growth Rules

The growth rules presented in reference 1 are formally sound and complete. Part of them also have a very straightforward correspondence with some of our rules. For the other part, namely the growth rules which produce two edges, there is no direct way of translating them into a 2-MCFG. Adding memory to the grammar by adding states that describe two neighbouring edges might seem appealing but would not solve the problem; two neighbouring edges 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. In fact, such a translation would be a guarantee of completeness.

4.3 Insights from promotion

The act of promotion translates to cyclic rotation of the A_2 web [Petersen]. Perhaps an interesting question here is whether promotion can be handled by a 2-MCFG (as a *context-free rewriting system*). If that is the case, 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 transcribing the solution to all other words belonging to the same orbit could then be a guideline towards completeness. The scale of such a task would of course be much greater, but the combination of these theoretical backbones into a unified body, combined with our mostly applied approach, could shed some light on the problem at hand.

5. Tools

6. Conclusion