Cooler than Cool:
Cool-Lex Order for Generating New Combinatorial Objects

Paul Lapey

May 9, 2022

Contents

1 Introduction **4**
- 1.1 Combinatorial Generation: Looking at All the Possibilities . . . 4
- 1.2 Lexicographic Orders . . . 4
  - 1.2.1 Lexicographic Sublists . . . 5
- 1.3 Binary Reflected Gray Code . . . 5
  - 1.3.1 Gray Code Sublists . . . 7
- 1.4 Cool-Lex Order . . . 7
  - 1.4.1 Cool-Lex Sublists . . . 8
- 1.5 Gray Codes for Trees and Catalan Objects . . . 9
- 1.6 Goals and Results . . . 9

2 Catalan Objects **11**
- 2.1 Dyck Words and Paths . . . 11
- 2.2 Binary Trees . . . 12
- 2.3 Ordered Trees . . . 12
- 2.4 Bijections . . . 14
  - 2.4.1 Binary Trees and Dyck Words . . . 14
  - 2.4.2 Ordered Trees and Dyck Words . . . 14
- 2.5 Minimal Changes . . . 15
- 2.6 Recursive Enumeration . . . 15

3 Cool-Lex Order **17**
- 3.1 Shifts and Non-Increasing Prefixes . . . 17
  - 3.1.1 Left-Shifts . . . 17
  - 3.1.2 Non-Increasing Prefixes . . . 17
- 3.2 (s,t)-Combinations: Fixed-Weight Binary Strings . . . 18
  - 3.2.1 Efficient Implementation . . . 18
- 3.3 Cool Lex Order on Dyck Words and Binary Trees . . . 19
  - 3.3.1 Efficient Implementation . . . 19
- 3.4 Multiset Permutations . . . 22
  - 3.4.1 Efficient Implementation . . . 23

4 The First Pop-Push Gray Code for Ordered Trees **25**
- 4.1 Relationship to Previous Results . . . 25
- 4.2 Terminology and Conventions . . . 25
- 4.3 Successor Rule . . . 26
- 4.4 Proof of Correctness . . . 27
  - 4.4.1 Definitions and Remarks . . . 27
  - 4.4.2 Correspondence between coolCat and nextree . . . 30

5 Loopless Ordered Tree Generation **34**
- 5.1 Relationship to Previous Results . . . 34
- 5.2 Storing Children in a Linked List . . . 35
- 5.3 Storing Children in an Array . . . 36

6 Lattice Paths: Łukasiewicz, Motzkin, Schröder **38**
6.1 Motzkin Paths . . . 38
6.2 Schröder Paths . . . 39
6.3 Łukasiewicz Paths . . . 39

7 A New Shift Gray Code for Łukasiewicz Words **41**
7.1 Successor Rule . . . 41
7.1.1 Observations . . . 41
7.2 Proof of Correctness . . . 41
7.2.1 Terminology and Remarks . . . 43
7.2.2 Equivalence . . . 44

8 Loopless Łukasiewicz and Motzkin Word Generation **45**
8.1 Generating Łukasiewicz Words in Linked Lists . . . 45
8.1.1 Implementation Observations . . . 45
8.2 Generating Motzkin Words in Arrays . . . 48

9 Final Remarks **49**
9.1 Summary . . . 49
9.2 Open Problems . . . 49

Chapter 1

Introduction

1.1 Combinatorial Generation: Looking at All the Possibilities

Combinatorial generation is defined as the exhaustive listing of combinatorial objects of various types. Frank Ruskey duly notes in his book *Combinatorial Generation* that the phrase "Let's look at all the possibilities" sums up the outlook of his book and the field as a whole [Rus03]. Examining all possibilities fitting certain criteria is frequently necessary in fields ranging from mathematics to chemistry to operations research. Combinatorial generation as an area of study seeks to find an underlying combinatorial structure to these possibilities and utilize it to obtain an algorithm to efficiently enumerate an appropriate representation of them [Rus03].

Combinatorial generation has many parallels to sorting. It is a fundamental computational task that will continue to be a necessary part of solving difficult problems for the foreseeable future. Additionaly, different sorting algorithms are often better suited for for different types of data. For example, bubble sort is generally slower than merge sort, but can be faster when the initial data is already close to being in sorted order. Similarly, different combinatorial generation algorithms may be better suited for different types of tasks. This makes combinatorial generation an important area of study for making many common tasks more efficient.

Unlike sorting, however, combinatorial generation is not widely taught and is rarely discussed in core textbooks. The foundational textbook series The Art of Computer proramming was initially released in 1968; combinatorial generation was not discussed until volume 4A, released in 2011 [Knu15]. Another notable combinatorial generation textbook is Kreher and Stinson's *Combinatorial Algorithms: Generation, Enumeration, and Search* [KS20]. Outside of these, however, discussion of combinatorial generation in textbooks is rare.

1.2 Lexicographic Orders

Lexicographic order is the simplest and most intuitive way of enumerating combinatorial objects. As its name suggests[1], it is the order typically used to order words in a dictionary. Any language of strings formed from a set of symbols with a total ordering has a lexicographic ordering. In particular, a lexicographic ordering is a total order of strings formed from symbols with a total ordering. An additional benefit of lexicographic ordering is that listing a set of combinatorial objects in lexicographic order is always possible. Any combinatorial object handled by computers must be encoded in binary somehow; therefore any combinatorial object can be enumerated in lexicographic order via the lexicographic order of its binary representations.

In addition to traditional lexicographic order, there are several closely related variations of it. Colexicographic order generates objects as if they were being generated via a lexicographic ordering of strings read in reverse. Where lexicographic order increments the rightmost bit and carries to the left,

[1] *Lexicography* is a word for the practice of compiling dictionaries. It draws its roots from the Greek λεξικος (lexikos), meaning "of words," and γραφη (graphe), meaning drawing or writing. Lexicography, therefore, can be thought of as writing about words. Interestingly, λεξικος and the English words *lecture*, *legend*, *legible*, and *legume*, all stem from the same Proto-Indo-European word meaning to collect. Reconstructing a common thread among these etymologically related words is left as an exercise to the reader.

co-lexicographic order increments the leftmost bit and carries to the right. Reverse lexicographic order generates objects in the reverse order of lexicographic order: it order objects from lexicographically largest to lexicographically smallest. Co-lexicographical order can also be reversed in this way to generate reverse co-lexicographic order. Figure 1.1 demonstrates lexicographic, co-lexicographic, and reverse lexicographic order for binary strings with n bits. Although all four lexicographic orderings are similar for binary strings, they are often very different from each other for for enumerating other sets of objects.

Continuing with the comparison of combinatorial generation to sorting, lexicographic and co-lexicographic orders are much like insertion and selection sort. They are fairly intuitive, easy to implement, and sometimes "good enough." However, they are rarely optimal: more thoughtful orderings will frequently have significant performance advantages over lexicographic orderings. In particular, lexicographic orderings often require worst-case $O(n)$ time per generated object, whereas *loopless* generation algorithms that use worst-case constant time per generated object are often achievable. The term loopless refers to the fact that, typically, a worst-case $O(1)$ algorithm can be implemented without any inner loops.

1.2.1 Lexicographic Sublists

An interesting observation is that lexicographic orderings for any fixed length binary language can be obtained by filtering the lexicographic ordering of all binary strings to obtain a sub-list. Specificaly, if a language $\mathcal{L}$ is a subset of n-bit binary strings, generate all n-bit binary strings and filter the list to contain only strings in $\mathcal{L}$. The resulting list will be a lexicoraphic ordering of the strings in $\mathcal{L}$. The same property holds for co-lexicographic order, reverse lexicographic order, and reverse co-lexicographic order. Figure 1.1 illustrates generating 4-bit binary strings and filtering that list to obtain a sublist of binary strings with 2 ones and 2 zeroes. It does this for lexicographic orderings, the binary reflected Gray code to be discussed in 1.3, and cool lex order, discussed in 1.4.

1.3 Binary Reflected Gray Code

A quintessential result of combinatorial generation in practice is Frank Gray's reflected binary code, or Gray code. The binary reflected Gray code gives a "reflected" ordering of binary strings such that each successive string in the ordering differs from the previous string by exactly one bit. This contrasts from a lexicographic ordering of binary strings, in which a n-digit binary string can differ by up to n digits from its predecessor and will differ by approximately two (more precisely $\sum_{i=0}^{n} \frac{1}{2}^i$, which is 1.9375 for 4 bit values and 1.996 for 8 bit values) bits on average[2]. The binary reflected Gray code, therefore, provides an ordering that requires half as many bit switches on average as the more intuitive lexicographic order.

The iterative successor rule for the binary reflected Gray code has 2 cases and is as follows:

Let α be a binary string and let f be the first bit of α that is equal to 1.

$$\text{gray}(\alpha) = \begin{cases} \text{complement}(1) & \text{if the parity of } \alpha \text{ is even} \\ \text{complement}(f+1) & \textit{otherwise} \end{cases} \tag{1.1}$$

Binary reflected Gray codes are especially useful in electromechanical switches to reduce physical error and prevent spurious output associated with asynchronous bit switches. In particular, changing multiple bits per iteration can result in "in-between" states where some but not all of the bit changes necessary for a switch have been executed. One can think of this like an odometer on a car: When changing from 99999 to 100000 miles, the odometer might briefly read 000000, or 19999, or 10009, or any number of other "in-between" states.

This issue can occur in cases as simple as incrementing 3 to 4. In lexicographic order, the string must change from 011 to 100; in Gray code order it changes from 010 to 110. The lexicographic change requires three bit changes; the Gray code order requires only one. When using physical switches, three bit changes are unlikely to change in exact synchrony. This creates the possibility of reading 101,

[2]Consecutive pairs of binary digits in lexicographic order will differ in the bit at position i with probability $\frac{1}{2}^i$. Therefore, the average number of differing bits between two binary strings of length n is $\sum_{i=0}^{n} \frac{1}{2}^i$, which converges to 2 as n grows large.

(a) The 2^4 4-bit binary strings generated lexicographic, co-lexicographic, reverse lexicographic, reverse co-lexicographic, binary reflected Gray code, and cool-lex order.

n	lex	colex	revlex	revcolex	Gray	cool
0	0000	0000	1111	1111	0000	0000
1	0001	1000	1110	0111	0001	1000
2	0010	0100	1101	1011	0011	1100
3	0011	1100	1100	0011	0010	1110
4	0100	0010	1011	1101	0110	1111
5	0101	1010	1010	0101	0111	0111
6	0110	0110	1001	1001	0101	1011
7	0111	1110	1000	0001	0100	1101
8	1000	0001	0111	1110	1100	0110
9	1001	1001	0110	0110	1101	1010
10	1010	0101	0101	1010	1111	0101
11	1011	1101	0100	0010	1110	0011
12	1100	0011	0011	1100	1010	1001
13	1101	1011	0010	0100	1011	0100
14	1110	0111	0001	1000	1001	0010
15	1111	1111	0000	0000	1000	0001

(b) 4-bit binary strings filtered to only contain strings with 2 ones and 2 zeroes

n	lex	colex	revlex	revcolex	Gray	cool
0						
1						
2					0011	1100
3	0011	1100	1100	0011		
4					0110	
5	0101	1010	1010	0101		
6	0110	0110	1001	1001	0101	
7						
8					1100	0110
9	1001	1001	0110	0110		1010
10	1010	0101	0101	1010		0101
11						0011
12	1100	0011	0011	1100	1010	1001
13						
14					1001	
15						

(c) Condensed version of the orders for binary strings with 2 ones and 2 zeroes obtained above

lex	colex	revlex	revcolex	Gray	cool
0011	1100	1100	0011	0011	1100
0101	1010	1010	0101	0110	0110
0110	0110	1001	1001	0101	1010
1001	1001	0110	0110	1100	0101
1010	0101	0101	1010	1010	0011
1100	0011	0011	1100	1001	1001

Figure 1.1: The 2^4 4-bit binary strings generated lexicographic, co-lexicographic, reverse lexicographic, reverse co-lexicographic, binary reflected Gray code, and cool-lex order.

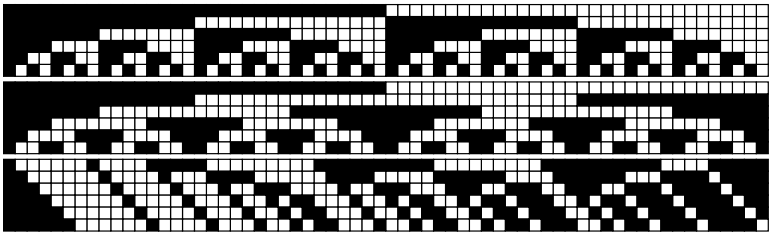


Figure 1.2: A visual representation of Lexicographic (top), binary reflected Gray code (middle), and cool-lex (bottom) enumerations of 6-bit binary strings. Individual strings are read vertically with the most significant bit at the top; white is 1.

110, 111, or *any other 3-bit binary number* if the switches are read mid-change, depending on the order of the bit changes. When using the binary reflected Gray code, the only states are $3 = 010$, the previous value, and $4 = 110$, the correct next value. One could easily imagine a test, such as checking if a number is less than 5, that could evaluate incorrectly due to reading during the change between $3 = 011$ and $4 = 100$ in lexicographic order. This type of error is eliminated by using a Gray code ordering that changes only one bit per iteration.

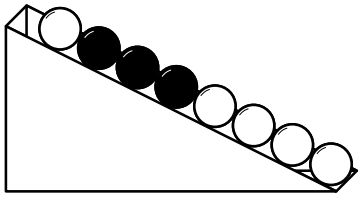
Frank Gray's binary reflected code was influential enough that the term *Gray code* is often used as a general term for any minimal change ordering of combinatorial objects. If lexicographic orderings with worst-case $O(n)$ time per generated object are like $O(n^2)$ sorting algorithms, Gray codes are like more efficient sorting algorithms such as merge sort, quick sort[3], or radix sort. Gray codes perform iterations by modifying a single object in place, not generating a new object from scratch. This is almost a hard requirement for achieving better than $O(n)$ time per object, as generating an object of size n from scratch is always at least $O(n)$.

1.3.1 Gray Code Sublists

In particular, Gray codes for many binary languages can be obtained by listing all binary strings in Gray code order, filtering the list to contain only strings in the language, and seeing if the resulting ordering is a Gray code. Sawada, Williams, and Wong demonstrated this process in practice for a broad set of binary "flip-swap languages" including necklaces, Lyndon words, and feasible 0-1 knapsack problem solutions, among others [SWW21]. They found that orderings obtained for binary "flip-swap" languages by filtering the binary reflected Gray code all have the property that successive strings differ from each other by at most 2 bits, allowing for simple and efficient implementations of these orderings.

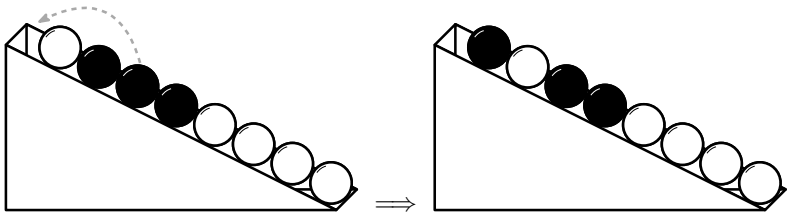
1.4 Cool-Lex Order

Cool-lex order can most easily be introduced using marbles on a ramp. Suppose one has n marbles on a ramp, with each marble colored black or white.

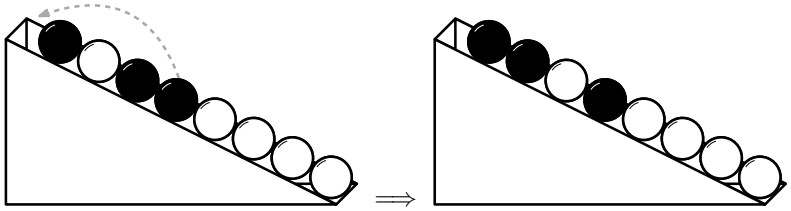


Let x be the first black marble that follows a white marble. Consider picking up and moving marble $x + 1$ to the top of the ramp. This yields a new configuration.

[3]Quicksort is actually worst-case $O(n^2)$, but has better average performance than simple $O(n \log(n))$ sorting algorithms like mergesort or heapsort. This is due to space efficiency and cache performance, among other things. Other sorting algorithms like introsort, pdqsort, and Timsort use hybrid approaches to obtain quicksort-like (or better) average performance while maintaining $O(n \log(n))$ worst-case runtime.



Now, consider performing the same move again: finding x, the first black marble that follows a white marble, and moving marble $x+1$ to the top of the ramp. This yields a third configuration.



Amazingly, repeatedly performing this operation will eventually generate all possible marble arrangements (with no distinction made between individual black and white marbles). To make the order cyclic, add an additional rule for the case where no black marble follows a white marble (all black marbles come before all white marbles). In this case, shift the last marble to the top of the ramp.

This rule for marbles on a ramp can be translated to binary strings to create a successor rule that generates all binary strings with fixed content. We refer to this operation of moving a single marble to the top of the ramp as a *left-shift*. The order described by this rule is cool-lex order, a variation of co-lexicographic order that was first introduced for (s,t)-combinations by Ruskey and Williams [RW05] [RW08]. It was initially discovered by thinking about simple iterative successor rules like the successor rule for the binary reflected Gray code in equation 1.1. To generate all binary strings of a given length instead of all binary strings with a fixed set of content, one only needs to change the rule slightly. In the case where there is no black marble following a white marble, move the last marble to the top of the ramp and change its color. In binary strings, this rule corresponds to left-shifting the last bit to the front of the string and complementing it.

Cool-lex order has some advantages of both lexicographic orderings and Gray codes. Like lexicographic orders, cool-lex order has a simple recursive structure that is easy to understand at high level. Like the binary reflected Gray code, cool-lex orders often lead to generation algorithms that are simpler and more efficient than lexicographic orders. Accordingly, cool-lex objects for exhaustively generating different sets are not just of theoretical interest. For example, the "multicool" package in R uses a loopless cool-lex algorithm to efficiently enumerate multiset permutations. The package started using cool-lex order for multiset permutations in version 1.1 and as of version 1.12 has been downloaded nearly a million times [CWKB21]. Moreover, due to its efficiency and simplicity, Don Knuth included the cool-lex algorithm for combinations in his 4th volume of *The Art of Computer Programming* and provided an implementation of it for his theoretical MMIX processor architecture due to its efficiency and simplicity [Knu15]. Additionally, a hardware implementation of the cool-lex order for combinations in a FPGA was introduced during Madeline Burbage's award winning ACM student research presentation [Bur20].

1.4.1 Cool-Lex Sublists

Like the binary reflected Gray code, cool-lex order has also been used to create additional minimal change orders via taking sublists. In particular, a binary *bubble language* is a set of binary strings with the closure property that the first 01 of any string can be replaced by 10 to obtain another string in th set. Ruskey, Sawada, and Williams found that all binary *bubble languages* appear in a Gray code order when listed in cool-lex order. Specifically, any binary bubble language can be listed in cool-lex order such that successive strings differ by one or two transpositions. The approach used in this paper is similar to that discussed in [SWW21] for generating Gray codes for "flip-swap" languages. A Gray code can be obtained for any *fixed weight* binary bubble language, or a binary bubble language that is a subset of (s,t)-combinations, by generating all (s,t)-combinations and filtering to contain strings in the relevant language. Gray codes for bubble languages of varying weights can be obtained

by concatenating fixed weight binary bubble language Gray codes together. Importantly, although this approach initially involves generating a superset of a language and filtering, careful analysis of the filtered order can typically yield a direct successor rule that efficiently generates only strings in the language without filtering. This has allowd for different versions of cool-lex order to efficiently enumerate Dyck words and binary trees [RW08], k-ary Dyck words, [DLM+12], and fixed-density necklaces and other languages [SW09].

In addition to the cool-lex order for (s,t)-combinations and bubble languages, cool-lex order has been generalized to enumerate other languages as well. In particular, cool-lex order for all binary strings of a given length [SW12] and multiset permutations [Wil09a] have been developed as well. Our result in Chapter 7 will provide a Gray code for a subset of multiset permutations in cool-lex order that was obtained initially by filtering the cool-lex order for multiset permutations. The resulting rule generates fixed-content Łukasiewicz words using one left-shift per generated string.

1.5 Gray Codes for Trees and Catalan Objects

This thesis aims to create efficient minimal change orderings for other objects, namely ordered trees and Łukasiewicz words. Ordered trees and Łukasiewicz words are both *Catalan objects*. The number of Łukasiewicz words of length $n+1$ and the number of ordered trees with $n+1$ nodes are both counted by the $n^{\underline{th}}$ Catalan number $\mathcal{C}_n$.

Frank Gray's reflected binary code used complementing a single bit as the minimal change between successive binary strings in its ordering. Other notions of minimal changes in strings are *adjacent-transpositions*, or *swaps*, which interchange two adjacent symbols in a string, and *shifts*, in which a single symbol in a string moved to another position. Łukasiewicz words are typically represented as strings of integers, and therefore can make use of these minimal change string operations. Our Gray codes for Łukasiewicz words will use a slightly more restrictive type of shift, a *left-shift*, which moves a single symbol somewhere to the left within a string.

Defining minimal changes for trees is more complicated, as what changes are *minimal* within a tree is often representation dependent. Our Gray code for ordered trees aims to use simple operations that are minimal for most reasonable tree representations. These minimal changes use "pops", which remove a node's first child, and "pushes," which push one node to be the first child of another. More specifically, the algorithm will generate trees using a pop-push operation that pops one node's first child and pushes it to become the first child of another node. We will refer to this pop-push operation as a "pull."

1.6 Goals and Results

This thesis has the broad goal of extending cool-lex order to new objects. It provides two primary contributions, each with sub-contributions.

The first contribution is a 'pop-push' Gray code for enumerating ordered trees in $O(1)$ time. Chapter 4 will give a two-case successor rule for generating all ordered trees with n nodes using at most two "pull" operations. Chapter 5 will provide a loopless algorithm for the successor rule in 4 and an implementation of the algorithm in C. This algorithm is an extension of the cool-lex order for Dyck words and binary trees presented by Ruskey and Williams. It shows that the cool-lex order for Dyck words is simultaneously a Gray code for a third Catalan object: ordered trees, as well as Dyck words and binary trees. These results have been submitted to an international algorithms conference and are currently under review [LW22b].

The second contribution is a shift Gray code for Łukasiewicz words. Łukasiewicz words are a generalization of Dyck words that allow for broader sets of content while maintining a notion of "balance." Chapter 7 will give a shift Gray code for generating Łukasiewicz words with fixed content using one prefix shift per iteration. Chapter 8 will give a loopless implementation of the algorithm in 7 using an array based implementation for the special case of Motzkin words and a linked list implementation for the general case of unrestricted Łukasiewicz words. These results have been accepted at the 33rd International Workshop on Combinatorial Algorithms and will be presented June 2022 [4] [LW22a].

[4] Although these results are presented after the ordered trees result in this paper, these results were developed earlier and therefore were submitted to an earlier conference

Chapter 2 will give background information on the Catalan numbers and their relation to the combinatorial objects generated by this thesis. Chapter 3 will give background information on existing applications of cool-lex order. Chapters 4 and 5 will give a new successor rule and loopless implementation for generating ordered trees in cool-lex order. Chapter 6.3 will give background information on generalizations of Dyck words, including Łukasiewicz words and Motzkin words. Chapters 7 and 8 will give a successor rule and loopless implementation for generating fixed-content Łukasiewicz words.

Chapter 2

Catalan Objects

The Catalan numbers are one of the most ubiquitous sequences of numbers in mathematics. Named for mathematician Eugene Charles Catalan, the $n^{\underline{th}}$ Catalan number can be succinctly defined as the number of ways of triangulating a convex polygon with $n+2$ sides. Figure 2.2a demonstrates this for the case of $n=3$. The sequence of Catalan numbers for $n \geq 0$ can be defined mathematically as follows:

$$\mathcal{C}_n = \frac{(2n)!}{n!(n+1)!} = 1, 1, 2, 5, 14, 42, 132, \ldots \qquad \text{OEIS}A000108[\text{ST20}] \qquad (2.1)$$

The Catalan numbers count a remarkable number of interesting and useful combinatorial objects that are in bijective correspondence with triangulations of n-gons. Combinatorial objects counted by the Catalan numbers are referred to as *Catalan objects*. Richard Stanley's book *Catalan Numbers* gives hundreds of examples of Catalan objects as well as a thorough history on the numbers and their study [Sta15]. This thesis will focus primarily on three Catalan objects: Dyck words, binary trees, and ordered trees. Sections 2.1, 2.2, and 2.3 consider Dyck words, binary trees, and ordered trees respectively. Section 2.4 discusses bijections between Dyck words, binary trees, and ordered trees; the bijection in 2.4.2 is particularly relevant to later results. Section 2.5 discusses minimal changes in the context of Catalan objects. Section 2.6 gives additional background on the recursive derivation of the Catalan numbers with examples using several different Catalan objects.

2.1 Dyck Words and Paths

The language of binary Dyck words is the set of binary strings that satisfy the following conditions: The string has an equal number of ones and zeroes and each prefix of the string has at least as many ones as zeroes. The number of distinct Dyck words with n ones and n zeroes is equal to the $n^{\underline{th}}$ Catalan nummber, $\mathcal{C}_n$. Dyck words with n ones and n zeroes are frequently referred to as Dyck words of *order* n. For example, the $\mathcal{C}_2 = 2$ Dyck words of order 2 are 1100 and 1010.

Two common interpretations of Dyck words are balanced parentheses and paths in the Cartesian plane. If each one in a Dyck word is taken to represent an open parenthesis and each zero a closing parenthesis, the Dyck language becomes the language of balanced parentheses. Alternatively, the Dyck language can be interpreted as the set of paths in the Cartesian plane using $(1,1)$ (northeast) and $(1,-1)$ (southeast) steps that start at $(0,0)$, end at $(0,0)$ and never go below the x axis. In this case, each one in a Dyck word represents a $(1,1)$ step and each zero represents a $(1,-1)$ step.

Figure 2.1 gives an illustration of each of these interpretations of Dyck words for $n=4$.

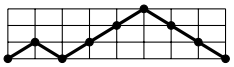
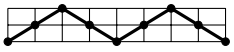
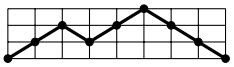
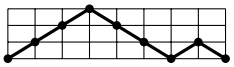
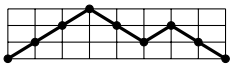
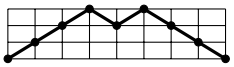
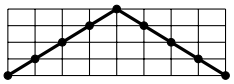
Dyck Path	Dyck Word	Parentheses
	10101010	()()()()
	10101100	()()(())
	10110010	()(())()
	10110100	()(()())
	10111000	()((()))
	11001010	(())()()
	11001100	(())(())
	11010010	(()())()
	11010100	(()()())
	11011000	(()((()))
	11100010	((()))()
	11100100	((())())
	11101000	(((())))
	11110000	(((())))

Figure 2.1: The $\mathcal{C}_4 = 14$ Dyck words of order 4 in lexicographic order

2.2 Binary Trees

Binary trees are fundamental objects in computer science, and are commonly used for searching, sorting, and storing data hierarchically. A binary tree can be defined recursively as follows: A binary tree either is the empty tree or 3-tuple (L, S, R), where S is the root of the tree, L is a left subtree, and R is a right subtree. A closely related object is an *extended binary tree*, which is a binary tree for which every non-leaf node has exactly two children.

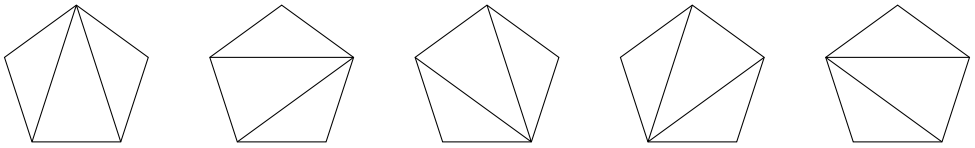
Binary trees and extended binary trees are both counted by the Catalan numbers: $\mathcal{C}_n$ is the number of binary trees with n nodes and the number of extneded binary trees with n internal nodes. A binary tree b with n nodes can be constructed from an extended binary tree e with n internal nodes by removing all leaves from the extended binary tree, leaving the n internal nodes as the only remaining nodes. This process can be reversed to construct an extended binary tree with n internal nodes from a binary tree with n nodes. Given a binary tree b with n nodes, add two leaf children to every leaf in b and add one leaf child to every node in b with one child. Following these steps, every node originally in b is now an internal node, and therefore the constructed tree is an extended binary tree with n internal nodes.

2.3 Ordered Trees

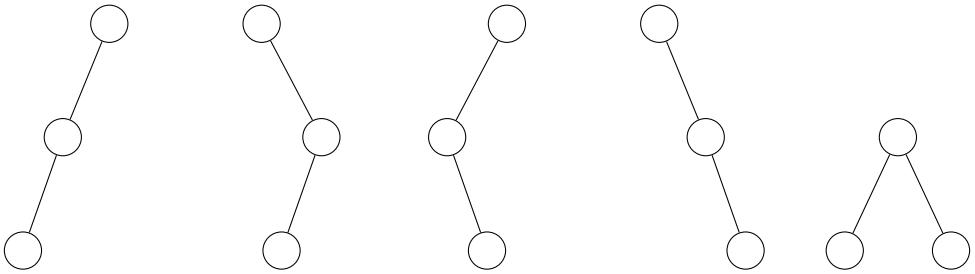
An ordered tree is a tree for which each node can have an unrestricted number of children and the order of a node's children is significant. An ordered tree can be defined recursively as follows:

An ordered tree is a tuple (r, C) where r is a root node and C is either the empty set ϕ or an ordered sequence of children $(P_1 \dots P_m)$ where each P_i is an ordered tree. Because of the designation of r as a root vertex, an ordered tree cannot be empty, unlike a binary tree.

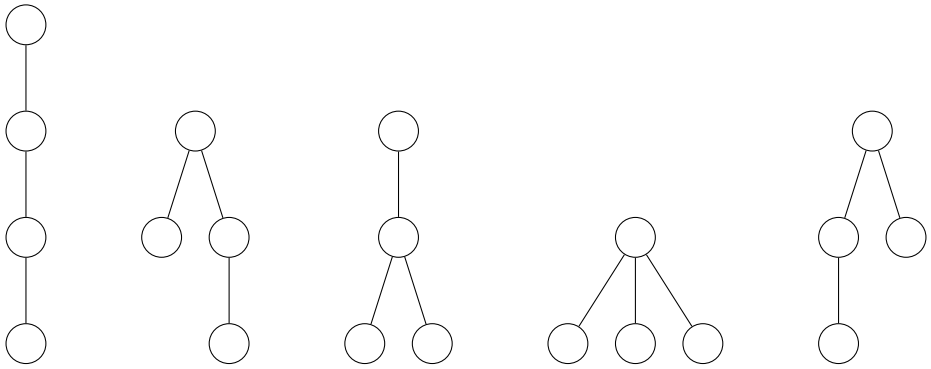
The number of ordered trees with $n + 1$ nodes is equal to $\mathcal{C}_n$.



(a) The $\mathcal{C}_3 = 5$ triangulations of a polygon with $3 + 2 = 5$ sides.



(b) The $\mathcal{C}_3 = 5$ binary trees with 3 nodes



(c) The $\mathcal{C}_3 = 5$ ordered trees with $3 + 1 = 4$ nodes

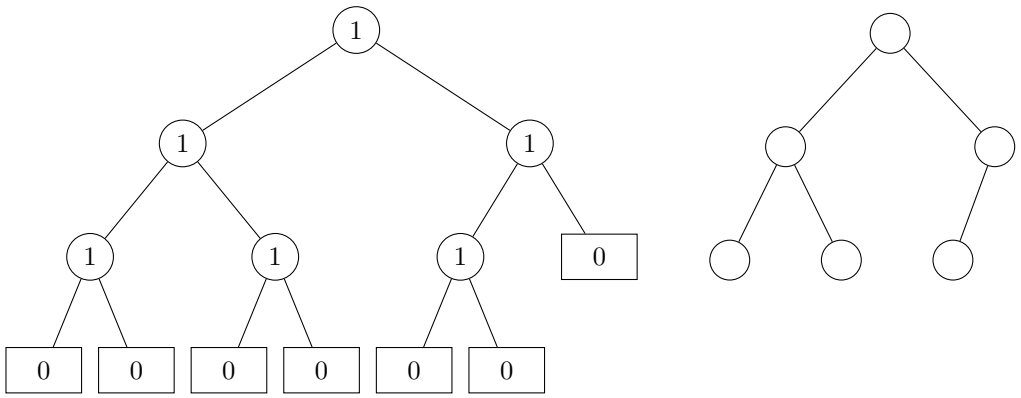


Figure 2.3: The extended binary tree (left) and binary tree (right) corresponding to the Dyck word 111001001100

Note that a preorder traversal of the extended binary tree excluding its final leaf yields 111001001100.

2.4 Bijections

Since Dyck words, binary trees, and ordered trees are all Catalan objects, all three sets of objects are in bijective correspondence with each other. For convenience, we will use the $\mathbf{D}_n$, $\mathbf{B}_n$, $\mathbf{E}_n$, and $\mathbf{T}_n$ to refer to the set of Dyck words of order n, the set of binary trees with n nodes, the set of extended binary trees with n internal nodes, and the set of ordered trees with $n+2$ nodes respectively. Section 2.4.1 discusses the bijection between Dyck words and binary trees; 2.4.2 discusses the bijection between Dyck words and ordered trees. It is interesting to note that the bijection between $\mathbf{D}_n$ and $\mathbf{E_n}$ is based on the nodes of the extended binary tree, while the bijection between $\mathbf{D}_n$ and $\mathbf{T}_n$ is based on the edges of the ordered tree.

2.4.1 Binary Trees and Dyck Words

The bijection between extended binary trees and Dyck words is particularly elegant: For any $e \in \mathbf{E}_n$, traverse e in preorder. Record a 1 for each internal node; record a 0 for each leaf ignoring the final leaf. The resulting binary sequence is a Dyck word $D \in \mathbf{D}_n$ corresponding to the extended binary tree e. This process can be reversed to go from $\mathbf{D}_n$ to $\mathbf{E}_n$. See figure 2.3 for an illustration of this process: traversing the extended binary tree on the right in preorder yields its corresponding Dyck word.

2.4.2 Ordered Trees and Dyck Words

The bijection between ordered trees and Dyck words is particularly relevant to this paper's results, as it is central to the loopless ordered tree generation algorithm given in Chapter 4. This algorithm will use the bijection between ordered trees and Dyck words specified in *Catalan Objects* [Sta15]. Figure 2.4 illustrates both directions of the bijection. The bijection can be formalized as follows:[1] Given an ordered tree T with $n+1$ nodes: Traverse T in preorder. Whenever going "down" an edge, or away from the root, record a 1. Whenever going "up" an edge, or towards the root, record a 0. The resulting binary sequence is a Dyck word D corresponding to the ordered tree T. This process can be inverted as follows: Let $D = d_1 ... d_{2n}$ be a Dyck word of order n with $n > 0$. Construct an ordered tree T via the following steps. Let T be an ordered tree with root r. Keep track of a current node c and set c equal to the root r.

- For each d_i such that $1 \leq i \leq 2n$
  - if $d_i = 1$, then add a rightmost child ch to c's children; set $c = ch$
  - if $d_i = 0$, then set c equal to c's parent.

Following the execution of these steps, r is the root of an ordered tree with n nodes corresponding to the Dyck word D.

[1] Stanley's text refers to ordered trees as *plane trees* and Dyck words as *ballot sequences*.

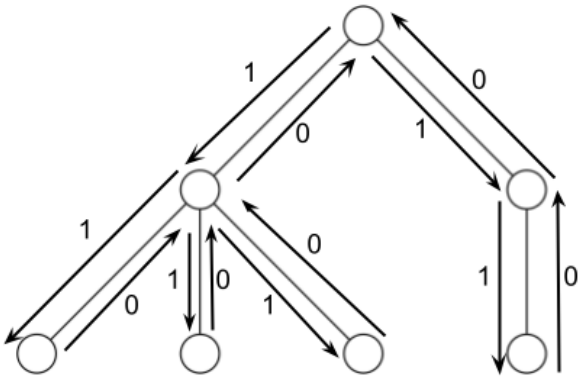


Figure 2.4: An ordered tree with $6 + 1 = 7$ nodes corresponding to the order 6 Dyck word 110101001100.

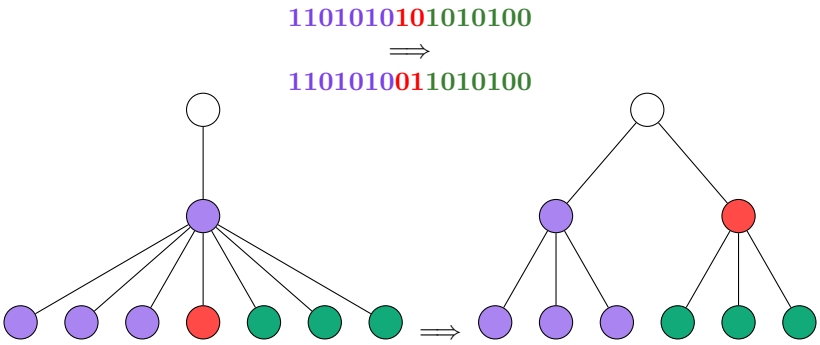


Figure 2.5: A single bit transposition in a Dyck word resulting in an ordered tree change that would require updating several pointers in a link-based representation. In particular, if this example were generalized to the case where the initial Dyck word is $1(10)^n0$, the corresponding ordered tree's first non-root node will have n children. In that case, transposing bits $n + 1$ and $n + 2$ in the Dyck word would require updating at least $n/2$ pointers in a link-based tree representation.

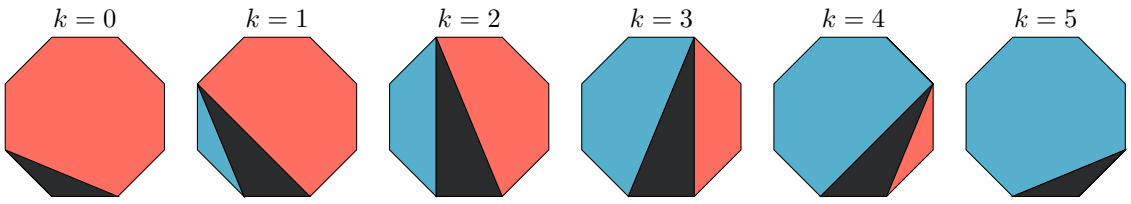
2.5 Minimal Changes

Much of this thesis's content concerns minimal change orderings for Catalan objects. However, our notion of a minimal change must differ for different Catalan objects. This is because a minimal change between two Catalan objects of one type will not necessarily correspond to a minimal change between two Catalan objects of a different type. Figure 2.5 gives an example where a single bit transposition in a Dyck word results in updating $O(n)$ nodes in an ordered tree. This makes the goal of creating an ordering that is a Gray code for multiple different combinatorial objects difficult, as it has to be a minimal change ordering for each object being generated.

We refer to orderings that are Gray codes for multiple combinatorial objects as *simultaneous Gray codes*. Ruskey and Williams showed that the cool-lex order for Dyck words is a simultaneous Gray code for Dyck words and binary trees [RW08]. Their result represented the first simultaneous Gray code between Dyck words and binary trees. More recently, additional simultaneous Gray codes have been discovered, including a simultaneous Gray code for Baxter permutations and rectangulations given by Hartung and Merino [HHMW20] [MM21]. In Chapter 4, we show that the cool-lex order for Dyck words and binary trees is also a Gray code for ordered trees, thus completing a single Gray-code ordering for three of the most important Catalan structures.

2.6 Recursive Enumeration

To gain more insight into Catalan objects, we conclude this chapter by considering an alternate formula for enumerating the Catalan objects and illustrate the formula for several specific objects.



$(\{\mathbf{D}_0\})\{\mathbf{D}_5\}$	$(\{\mathbf{D}_1\})\{\mathbf{D}_4\}$	$(\{\mathbf{D}_2\})\{\mathbf{D}_3\}$	$(\{\mathbf{D}_3\})\{\mathbf{D}_2\}$	$(\{\mathbf{D}_4\})\{\mathbf{D}_1\}$	$(\{\mathbf{D}_5\})\{\mathbf{D}_0\}$
2-gon; 7-gon	3-gon; 6-gon	4-gon; 5-gon	5-gon; 4-gon	6-gon; 3-gon	7-gon; 2-gon
$\mathcal{C}_0 \cdot \mathcal{C}_5$	$\mathcal{C}_1 \cdot \mathcal{C}_4$	$\mathcal{C}_2 \cdot \mathcal{C}_3$	$\mathcal{C}_3 \cdot \mathcal{C}_2$	$\mathcal{C}_4 \cdot \mathcal{C}_1$	$\mathcal{C}_5 \cdot \mathcal{C}_0$
$1 \cdot 42$	$1 \cdot 14$	$2 \cdot 5$	$5 \cdot 2$	$14 \cdot 1$	$42 \cdot 1$
42	14	10	10	14	42

$$C_6 = \sum_{k=0}^{5} \mathcal{C}_k \cdot \mathcal{C}_{n-k} = 42 + 14 + 10 + 10 + 14 + 42 = 132$$

Figure 2.6: Constructing $\mathcal{C}_6$ recursively using polygons and Dyck words. In the case of triangulated polygons, the blue region is a polygon with $k+2$ sides, the black region is a triangle, and the red region is a polygon with $n-k+2$ sides. The blue region can be triangulated $\mathcal{C}_k$ ways, the black region is already a triangle and therefore can be triangulated 1 way, and the red region can be triangulated C_{n-k} ways. Note again the special case of $k = 0$ or $n - k = 0$, where we define a 2-gon as having one triangulation, so $C_0 = 0$ In the case of Dyck words, each $\{\mathbf{D}_i\}$ can expand to any of the $\mathcal{C}_i$ Dyck words of order i. Thus, the expression $(\{\mathbf{D}_k\})\{\mathbf{D}_{n-k}\}$ has $\mathcal{C}_k \cdot \mathcal{C}_{n-k}$ possible expansions. Note that we define $\mathbf{D}_0$ as having only one possible expansion, the empty string.

The Catalan numbers can also be expressed through a summation that hints at the recursive structure of many of the Catalan objects. The recursive formula appears below, with $\mathcal{C}_0 = 1$.

$$\mathcal{C}_{n+1} = \sum_{k=0}^{n} \mathcal{C}_k \cdot \mathcal{C}_{n-k} \tag{2.2}$$

In the case of triangulated polygons, this summation can be derived as follows:

$\mathcal{C}_{n+1}$ is the number of ways of triangulating a convex polygon with $n+3$ sides. Let $\mathcal{P}_{n+3}$ be a convex $(n+3)$-gon and let $\mathcal{T}$ be a triangulation of $\mathcal{P}_{n+3}$. Fix an edge e in $\mathcal{P}_{n+3}$. Note that e lies between two vertices of $\mathcal{P}_{n+3}$. e must be in exactly one triaingle in $\mathcal{T}$. Let T_i be the triangle in $\mathcal{T}$ that contains e. T_i must have two of its vertices on the two vertices of e and one vertex that is another vertex in $\mathcal{P}_{n+3}$. There are $n+1$ other vertices of $\mathcal{P}_{n+3}$. Suppose the third vertex of T_i is $k+1$ vertices clockwise of e, where $0 \leq k \leq n$. Drawing T_i divides $\mathcal{P}_{n+3}$ into 3 polygons: T_i, a $(k+2)$-gon clockwise of T_i, and a $(n-k+2)$-gon counterclockwise of T_i. This means that for each possible value of k, there is one way of triangulating T_i, $\mathcal{C}_k$ ways of triangulating the polygon clockwise of T_i, and $\mathcal{C}_{n-k}$ ways of triangulating the polygon counterclockwise of T_i. Therefore, there are $C_k \cdot C_{n-k}$ ways of triangulating $\mathcal{P}_{n+3}$ for each value of k. Therefore, there are $\mathcal{C}_{n+1} = \sum_{k=0}^{n} \mathcal{C}_k \mathcal{C}_{n-k}$ total ways of triangulating $\mathcal{P}_{n+3}$. Figure 2.6 illustrates this process for the case of $n + 1 = 6$.

Chapter 3

Cool-Lex Order

This chapter will give background information on cool-lex order and the sets it has been used to exhaustively generate. Section 3.1 will discuss definitions and terminology relevant to the cool-lex algorithms discussed in this chapter. Section 3.2 will discuss the cool-lex algorithm for generating (s,t)-combinations. Section 3.3 will discuss generating Dyck words and binary trees in cool-lex order. 3.3 is of particular relvance to the result for generating ordered trees in 4. Section 3.4 discusses generating multiset permutations in cool-lex order. Figure 3.2 illustrates all three of these orders. Each section will also discuss efficient implementations of these algorithms. Specifically, each of these orders can be implemented looplessly with a careful implementation.

3.1 Shifts and Non-Increasing Prefixes

Two common threads in the cool-lex algorithms for combinatorial generation is their focus on the *non-increasing prefix* of string and their use of *left-shifts* for minimal changes. This section will define these terms and discuss their relevance to cool-lex order.

3.1.1 Left-Shifts

Mentioned previously in section 1.4, a left-shift shifts a single symbol somewhere earlier (left) in a string. Given a string $\alpha = a_1 a_2 \dots a_n$, we define $\text{left}_\alpha(i,j)$ with $i < j$ as removing the symbol α_j and inserting it at position i, shifting symbols α_i through α_{j-1} one position to the right. A formal definition of $\text{left}_\alpha(i,j)$ is given below in equation 3.1.

$$\text{left}_\alpha(i,j) = a_1 a_2 \dots a_{i-1} a_j a_i a_{i+1} \dots a_{j-1} a_{j+1} a_{j+2} \dots a_n. \tag{3.1}$$

Note that the left-shift operation preserves the set of content comprising α. Specifically, it reorders symbols within α but does not remove symbols from α or add any new symbols to α. As a simple example, performing $\text{left}(2,4)$ on the string 321$\underline{0}$21 shifts the underlined zero to position 2 and therefore yields 302121.

Cool-lex orders frequently left-shift a single symbol to the front (position 1) of the string. Thus, we also define a simplified left-shift operation which left-shifts a single symbol to position 1.

$$\text{left}_\alpha(j) = \text{left}_\alpha(1,j) \tag{3.2}$$

We also define a prefix-shift operation that shifts a single symbol into the second position. This operation will be used in section 3.3 for generating Dyck words.

$$\mathsf{preshift}_\alpha(j) = \text{left}_\alpha(2,j) \tag{3.3}$$

3.1.2 Non-Increasing Prefixes

Cool-lex algorithms shift symbols shortly after a string's non-increasing prefix. A non-increasing prefix is simply the longest prefix of a string such that each symbol is no larger than the previous symbol in

the string. We define the the the non-increasing prefix of α as $\alpha_1, \alpha_2, \cdots \alpha_m$ where m is the maximum value such that $\alpha_i \leq \alpha_{i+1}$ for all $1 \leq i < m$. Note that $\alpha_m < \alpha_{m+1}$.

The non-increasing prefix of a binary string is always 1^a0^b for some $0 \leq a, b$, where exponentiation denotes repeated symbols. This property will be especially convenient for cool-lex rules for binary langues in sections 3.2 and 3.3 as it will allow for implementations of left-shifts that transpose a constant number of bits instead of shifting an arbitrary number of symbols.

3.2 (s,t)-Combinations: Fixed-Weight Binary Strings

Recall the marble enumeration rule from 1.4. That rule can equivalently be used to generate all binary strings with a fixed set of content. Binary strings with s zeroes and t ones are often referred to as (s,t)-combinations, since each string can be used to represent a choice of t elements from a set of size of $s+t$. The cool-lex successor rule for generating all fixed-weight binary strings was given by Aaron Williams in his Ph. D thesis [Wil09c] and is the same as the aforementioned rule for marbles.

Let α be a binary string of length n. Let y be the position of the leftmost zero in α and x be the position of the leftmost 1 in α such that $x > y$. Note that x is the first 1 that follows a 0; the analogue of marble x in section 1.4. Moreover, $\alpha_1 ... \alpha_{x-1}$ is the non-increasing prefix of α.

$$\overleftarrow{\text{cool}}(\alpha) = \text{left}_\alpha(x+1) \tag{3.4}$$

To make this order cyclic, an additional case can be added for when there is no 1 following a zero. In this case, shift the last marble to the front of the list. Note that in this case all ones must follow all zeroes, and therefore $\alpha = 1^t0^s$. Figure 3.2a demonstrates this rule being used to enumerate $(4,2)$ combinations.

$$\overleftarrow{\text{cool}}(\alpha) = \begin{cases} \text{left}_\alpha(n) & \text{if } \alpha = 1^t0^s \\ \text{left}_\alpha(x+1) & otherwise \end{cases} \tag{3.5}$$

3.2.1 Efficient Implementation

At first glance, $\text{left}_\alpha(x+1)$ seems like an $O(n)$ opeartion, as x can be up to the length of α and the left-shift operation requires shifting each symbol from α_1 to α_x down. However, the property that the first x symbols of the string are non-increasing allows for simplification of the left-shift operation. Note that since α is a binary string, the non-increasing prefix of α must be 1^a0^b for some $0 \leq a, b$, as previously mentioned in 3.1. Since y is the index of the first 0 in α and x is the index of the first 1 following a 0 in α, we can conclude that $\alpha_1 ... \alpha_{x-1}$ must be exactly $1^{y-1}0^{x-y}$. This allows each left-shift operation can be replaced with with either one or two symbol transpositions

Let $\mathsf{transpose}(\alpha, i, j)$ with $1 \leq i \leq j \leq n$ be a function that swaps α_i an α_j. More formally, $\mathsf{transpose}(\alpha, i, j) = \alpha_1, \alpha_2, \ldots, \alpha_{i-1}, \alpha_j \alpha_{i+1} \ldots \alpha_{j-1} \alpha_i \alpha_{j+1} \ldots \alpha_n$. Equation 3.6 re-states the successor rule for combinations using transposes, with separte cases for whether a 1 or a 0 is left-shifted.

$$\overleftarrow{\text{cool}}(\alpha) = \begin{cases} \mathsf{transpose}(\alpha, y, x) & \text{if } \alpha_{x+1} = 1 \\ \mathsf{transpose}(\mathsf{transpose}(\alpha, y, x), 1, x+1) & otherwise \end{cases} \tag{3.6}$$

Moreover, scanning the string to find values for x and y is not necessary. Recall that $\alpha_{x-1} < \alpha_x$. Since α_{x+1} is always being shifted, the increase from α_{x-1} to αx is maintained, but shited one position to the right. Therefore, if $\alpha_{x+1} = 0$ and $\alpha_1 = 1$, the new value of x is 2, as the 1 at index 2 follows a zero. Otherwise, if $\alpha_{x+1} = 1$ or $\alpha_{x+1} = 0$ and $\alpha_1 = 0$, x can be incremented by one to obtain the value of x for the next string. Similarly, y is incremented by 1 if $\alpha_{x+1} = 1$ and set to 1 if $\alpha_{x+1} = 0$

Figure 3.1 gives psuedocode for implementing this successor rule looplessly.

```
coolCombo(s, t)
n ← s + t
b ← 1^t 0^s
x ← t
y ← t
visit(b)
while x < n do
    b_x = 0
    b_y = 1
    x ← x + 1
    y ← y + 1
    if b_x = 0 then
        b_x ← 1
        b_1 ← 0
        if y > 2 then
            x ← 2
        y ← 1
    visit(b)
```

Figure 3.1: Pseudocode for generating (s,t)-combinations in cool-lex order.

3.3 Cool Lex Order on Dyck Words and Binary Trees

Ruskey and Williams found the following algorithm for generating binary Dyck words, dubbed "CoolCat" due to its use of a cool-lex order to generate (cat)alan objects [RW08]. As in chapter 2, we will use $\mathbf{D}_n$ to denote binary Dyck words with n ones and n zeroes. Note that the length of any string in $\mathbf{D}_n$ is therefore $2n$. Let $D \in \mathbf{D}_n$. Recall that definition of the prefix shift operation from equation 3.3: $\mathsf{preshift}_D(j) = \mathrm{left}_D(2, j)$. Let y be the index of the first 0 in D and x be the index of the first 1 following a 0 in D as in section 3.2. Equation 3.7 gives the successor rule for Dyck words.

$$\overleftarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}_D(x+1) & \text{if } \mathsf{preshift}_D(x+1) \in \mathbf{D}_n \quad (3.7a) \\ \mathsf{preshift}_D(x) & \text{otherwise} \quad (3.7b) \end{cases}$$

Figure 3.2b shows this successor rule being used to geneate Dyck words of order 4.

Ruskey and Williams's algorithm can also enumerate a broader set of strings: The algorithm enumerates any set $\mathbf{D}_{s,t}$ where any $D \in \mathbf{D}_{s,t}$ has s zeroes and t ones and satisfies the constraint that each prefix of D has as many ones as zeroes. This is broader than the language of Dyck words, as it does not have the requirement that a string have an equal number of ones and zeroes. We will focus on $\mathbf{D}_n$ languages due to their correspondce with Dyck words and therefore other Catalan objects.

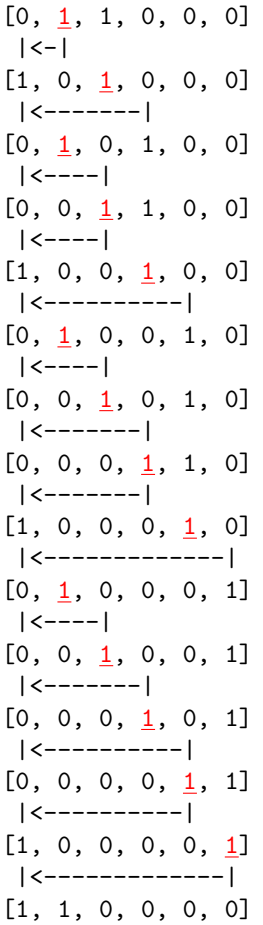
3.3.1 Efficient Implementation

Evaluating whether $\mathsf{preshift}_D(x+1) \in \mathbf{D}_n$ can be determined by looking at D_{x+1} and the the first $x-1$ symbols of D:

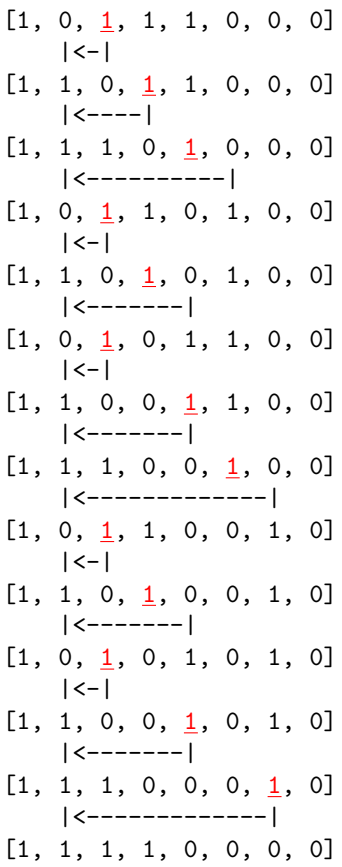
If $D_{x+1} = 1$, then shifting it into the second position is valid. If $D_{x+1} = 0$ and D starts with at least $\frac{x-1}{2}$ ones, then shifting a 0 into the second position will not invalidate the condition that all prefixes of D have at least as many ones as zeroes. Therefore, the successor rule in 3.7 can be simplified to the following:

$$\overleftarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}_D(x+1) & D_{x+1} = 1 \text{ or } D \text{ starts with at least } \lfloor \frac{x-1}{2} \rfloor \text{ ones} \quad (3.8a) \\ \mathsf{preshift}_D(x) & \text{otherwise} \quad (3.8b) \end{cases}$$

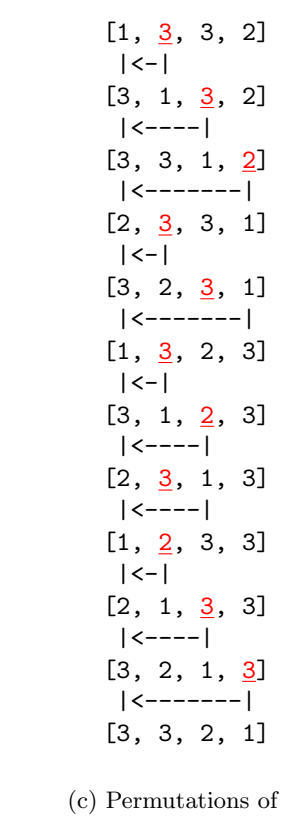
$\mathsf{preshift}_D(x+1) \in \mathbf{D}_n \iff D$ starts with more than $\lfloor \frac{x-1}{2} \rfloor$ ones



(a) $(s-t)$-combinations with $s = 4$ zeroes and $t = 2$ ones



(b) Dyck words of order 4



(c) Permutations of the multiset {1,2,3,3}

Figure 3.2: Cool-lex order for (s,t)-combinations, Dyck words, and multiset permutations. The symbol immediately following each string's non-increasing prefix, or the *first increase* of each string is highlighted in red. The arrows beneath each string show the shift that translates each string to its successor. Note that in all cases, either the first increase or the symbol immediately following it is left-shifted. For (s,t)-combinations and multiset permutations, symbols are always shifted to position 1. For Dyck words, symbols are always shifted to position 2.

Ruskey and Williams provided a loopless pseudocode implementation of CoolCat that utilized this fact to enumerate any $\mathbf{D}_{s,t}$ using at most 2 conditionals per successor [RW08]. Using the bijection between Dyck words and binary trees, Ruskey and Williams also showed that their successor rule can be translated to a loopless algorithm for generating all binary trees with n nodes.

An an additional simplifying observation is that all Dyck words start with 1. Thus, a conceivable representation of a Dyck word of order n would be a binary string of length $2n-1$, where the $2n-1$ bits of the representation correspond to bits 2 through $2n$ of the actual Dyck word, and the leading 1 is implied. In this case, each $\mathsf{preshift}_{\lceil}(D)]i$ can be replaced with $\text{left}_D(i)$, each shift now shifts a symbol to index 1.

With D redefined as above and x defined as before with respect to the updated representation of D, the successor rule for Dyck words can be further simplified to the following:

$$\overleftarrow{\mathsf{coolCat}}(D) = \begin{cases} \text{left}_D(x+1) & D_{x+1} = 1 \text{ or } D \text{ starts with at least } \lfloor \frac{x-2}{2} \rfloor \text{ ones} \quad (3.9a) \\ \text{left}_D(x) & \text{otherwise} \quad (3.9b) \end{cases}$$

Similarly to the rule for combinations in 3.6, the left-shifts in $\overleftarrow{\mathsf{coolCat}}$ can be implemented looplessly using transposes. Moreover, values for x and y can be kept track of based on which values are shfited, like in the (s,t) combinations algorithm. Since $y-1$ is the number of consecutive ones to start D, keeping track of y and x is sufficient to evaluate the condition in 3.9a. Pseudocode for implementing $\overleftarrow{\mathsf{coolCat}}$ looplessly is given in figure 3.3.

```
coolDyck(t)
n ← 2 · t
b ← 1^t 0^t
x ← t
y ← t
visit(b)
while x < n do
    b_x = 0
    b_y = 1
    x ← x + 1
    y ← y + 1
    if b_x = 0 then
        if x ≥ 2 · y − 2 then
            x ← x + 1
        else
            b_x ← 1
            b_2 ← 0
            x ← 3
            y ← 2
    visit(b)
```

Figure 3.3: Pseudocode for generating Dyck words in cool-lex order.

Due to its simplicity and efficiency, Don Knuth included the cool-lex algorithm for Dyck words in his 4th volume of *The Art of Computer Programming* and also provided an implementation of it for his theoretical MMIX processor architecture [Knu15]. Figure 3.4 gives demonstrates the iteration of coolCat on Dyck words of order 4.

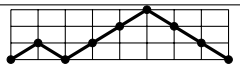
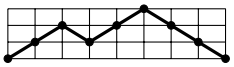
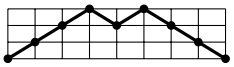
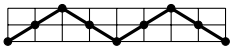
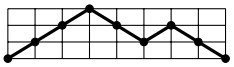
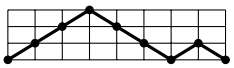
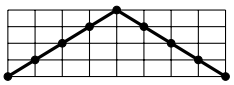
Dyck Path	Dyck Word	Parentheses
	10111000	()((()))
	11011000	(()(()))
	11101000	((()()))
	10110100	()(()())
	11010100	(()()())
	10101100	()()(())
	11001100	(())(())
	11100100	((())())
	10110010	()(())()
	11010010	(()())()
	10101010	()()()()
	11001010	(())()()
	11100010	((()))()
	11110000	(((())))

Figure 3.4: The $\mathcal{C}_4 = 14$ Dyck words of order 4 in cool-lex order

3.4 Multiset Permutations

Cool-lex order has also been shown to enumerate multiset permutations via prefix shifts. The rule given by Williams can be described via a simple algorithm that uses one left-shift per generated string [Wil09a].

First, recall equation 3.5 for (s,t)-combinations. Equation 3.10 gives an expanded version of the successor rule for (s,t)-combinations that differentiates between whether a 0 or a 1 is being shifted. Note that left-shifting position x and position $x+1$ are equivalent when $\alpha_{x+1} = 1$, since α_x is the first one that follows a zero and therefore $\alpha_{x+1} = \alpha_x = 1$.

$$\overleftarrow{\text{cool}}(\alpha) = \begin{cases} \text{left}_\alpha(n) & \text{if } \alpha = 1^t0^s \\ \text{left}_\alpha(x) & \text{if } \alpha_{x+1} = 1 \\ \text{left}_\alpha(x+1) & \textit{otherwise} \end{cases} \tag{3.10}$$

Let α be a multiset of length n. Let x be the smallest value for which $\alpha_x > \alpha_{x-1}$. Note that this makes $\alpha_1 \cdots \alpha_{x-1}$ the non-increasing prefix of α. The following successor rule generalizes 3.10 to generate all permutations of the content of the multiset α. See Figure 3.2c for an illustration of the shifts excuted by this successor rule and Figure 3.6 for a comparison of cool-lex order to lexicographic order for two multisets.

$$\text{nextPerm}(\alpha) = \begin{cases} \text{left}_\alpha(n) & \alpha \text{ is in descending order} \\ \text{left}_\alpha(x) & \alpha_{x+1} > \alpha_{x-1} \\ \text{left}_\alpha(x+1) & \textit{otherwise} \end{cases} \tag{3.11}$$

3.4.1 Efficient Implementation

This successor rule has the convenient property of ensuring that length of the successor's non-increasing prefix is easy to find. In particular, if α_{i+2} is shifted, then the length of the non-increasing prefix is either 1 if $\alpha_{i+2} \leq \alpha_1$ or $i+1$ otherwise. Similarly, if α_{i+1} is shifted, then the length of the non-increasing prefix is either 1 if $\alpha_{i+1} \leq \alpha_1$ or $i+1$ otherwise. This property makes a loopless implementation of the sucessor rule possible, as scanning the string to find the length of the non-increasing prefix is not required. A similar property regarding the length of successive non-increasing prefixes will allow for a loopless implementation of the shift Gray code for Łukasiewicz words in chapter 8.

However, unlike the binary cool-lex rules, multiset permutations can have any number of distinct symbols in a non-increasing prefix. Therefore, implementing a left-shift via transpositions would be more complicated and not a constant time operation. Therefore, [Wil09a] used a linked-list representation of a multiset to implement the successor rule in 3.11 looplessly.

```
function MULTICOOL(E)
    [h, i, j] ← init(E)
    visit(h)
    while j.n ≠ φ or j.v < h.v do
        if j.n ≠ φ and i.v ≥ j.n.v then
            s ← j
        else
            s ← i
        t ← s.n
        s.n ← t.n
        t.n ← h
        if t.v < h.v then
            i ← t
        j ← i.n
        h ← t
        visit(h)
```

Figure 3.5: Visits the permutations of multiset E. The permutations are stored in a singly-linked list pointed to by head pointer h. Each node in the linked list has a value field v and a next field n. The init(E) call creates a singly-linked list storing the elements of E in non-increasing order with h, i, and j pointing to its first, second-last, and last nodes, respectively. After the first iteration, i points to the last node in the multiset permutation's non-increasing prefix and j = i.n. Variables s and t are also pointer variables and are used for performing each shift. The null pointer value is given by ϕ. Note: If E is empty, then init(E) should exit. Also, if E contains only one element, then init(E) does not need to provide a value for i.

Due to the simplicity and efficiency of this rule, it is used in the "multicool" package in R, which is used for generating multiset permutations, Bell numbers, and other combinatorial objects [CWKB21] [**?**]

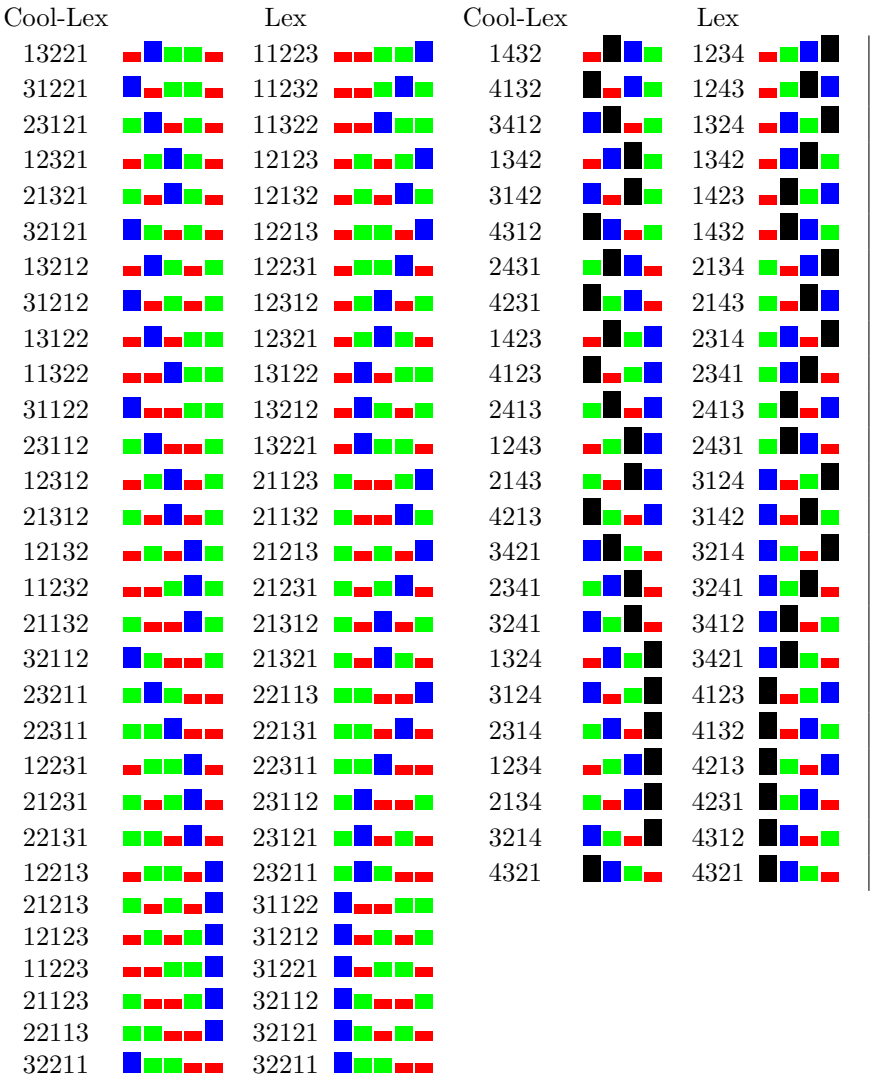


Figure 3.6: Illustration comparing cool-lex and lexicographic order for permutations of the multisets with content {1,1,2,2,3} and {1,2,3,4}

Chapter 4

The First Pop-Push Gray Code for Ordered Trees

This chapter presents a loopless Gray code for generating all ordered trees with n nodes. Sections 4.1 and 4.2 give context on this Gray code's relationship to related results and discuss terminology for this chapter's results. Section 4.3 provides a 2-case successor rule for the algorithm; Section 4.4 gives a proof of our algorithm's correctness.

4.1 Relationship to Previous Results

Ruskey and Williams previously gave a Gray code for generating all Dyck words of a given length in cool-lex order via prefix shifts [RW08]. In the same paper, Ruskey and Williams also gave a Gray code for generating all binary trees with a fixed number in the same order, making the cool-lex order of Dyck words a simultaneous Gray code for Dyck words of order n and binary trees with n nodes.

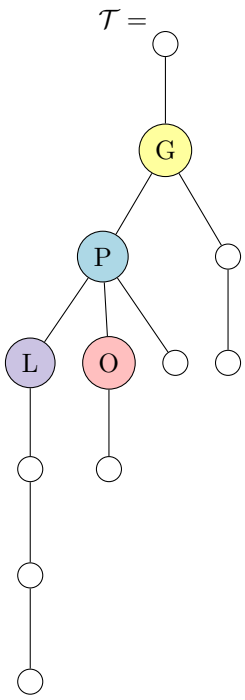
This thesis provides a new Gray code that generates another Catalan object, ordered trees, with a fixed number of nodes in a cool-lex order using *pops* and *pushes*. These pop and push operations remove a node's first child (pop) and push a node to be another node's new first child (push). Our Gray code generates all ordered trees using at most 2 pushes and 2 pops per generated tree. This constitutes the first pop-push Gray code for ordered trees. Additionally, the algorithm generates a minimal change ordering of ordered trees in the same order as their corresponding Dyck words in Ruskey and Williams's algorithm for Dyck words and binary trees. Therefore, in conjunction with the result from [RW08], the cool-lex order for Dyck words now creates a simultaneous Gray code for Dyck words, binary trees, and ordered trees. As we will see in chapter 5, this Gray code can be implemented via a looples algorithm like the the cool-lex algorithms for Dyck words and binary trees.

4.2 Terminology and Conventions

Throughout this chapter, we typeset ordered trees as $\mathcal{T}$, and we use n to denote the number of nodes in a tree. We typeset nodes of a tree as X, and we frequently refer to a node and the subtree rooted at that node interchangeably. Code is written with a `monospaced` font.

When discussing an ordered tree $\mathcal{T}$, we use the term *first branching*, which could be misinterpreted. We use the term to mean the following: The first branching occurs when the preorder traversal goes up an edge, and then down an edge, for the first time. The node that is incident to both of these edges is the parent of the first branching. In particular, note that the subtree to the left of the first branch must be a path (see Remark 4.3). Moreover, note that the first branching is also the first time a second child is visited in a preorder traversal of $\mathcal{T}$. All previously visited nodes in a preorder traversal are either the root or the first child of their parent.

Another way of describing the first branching is as follows. Let O be the first second-child node that is visited during a preorder traversal. If P is the parent of O, then P is the parent of the first branching. We frequently label the following nodes around the first branching: O and P are defined as above; L is P's first child, and G is P's parent. Fig. 4.1 gives an example illustrating an ordered



$$D = [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0]$$

Figure 4.1: An ordered tree with 12 nodes corresponding to the Dyck word 111111000011001011000. The *first branching* of $\mathcal{T}$ is from P to O.

tree's first branching and the nodes O, P, G, and L. In the special case that $\mathcal{T}$ has no first branching, and therefore $\mathcal{T}$ is a path, we define O to be $\mathcal{T}$'s unique leaf.

4.3 Successor Rule

Given an ordered tree $\mathcal{T}$ and an ordered tree node A in $\mathcal{T}$, let $\mathsf{popchild}(A)$ be a function that removes and returns A's first child. In other words, it pops A's first child. Additionally, let $\mathsf{pushchild}(A, B)$ be a function that makes B A's first child. In other words, it pushes B onto A's list of children. For convenience, we will also define $\mathsf{pull}(B, A) = \mathsf{pushchild}(A, \mathsf{popchild}(B))$, which removes the first child of A and makes it the new first child of B.

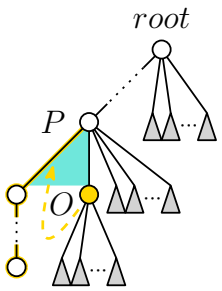
The successor rule for enumerating ordered trees with n nodes can be stated as follows:

$$\mathsf{nextree}(\mathcal{T}) = \begin{cases} \mathsf{pull}(O, P) & O \text{ has at least 1 child or } P = root \quad (4.1a) \\ \mathsf{pull}(G, P); \mathsf{pull}(root, P) & \text{otherwise} \quad (4.1b) \end{cases}$$

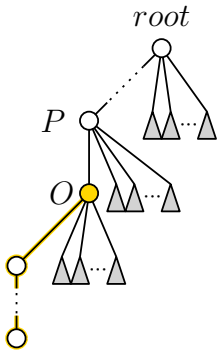
To make the order cyclic generated by (4.1) cyclic, an additional rule can be added for the case where $\mathcal{T}$ has no first branching, modifying the successor rule to be:

$$\mathsf{nextree}(\mathcal{T}) = \begin{cases} \mathsf{pull}(root, P) & \mathcal{T} \text{ has no first branching} \quad (4.2a) \\ \mathsf{pull}(O, P) & O \text{ has at least 1 child or } P = root \quad (4.2b) \\ \mathsf{pull}(G, P); \mathsf{pull}(root, P) & \text{otherwise} \quad (4.2c) \end{cases}$$

Figure 4.2 demonstrates of how (4.1a) and (4.1b) operate on trees in general. Figure 4.3 demonstrates the pull shifts from equations (4.1) and (4.2) as executed on four specific trees and their corresponding Dyck words.

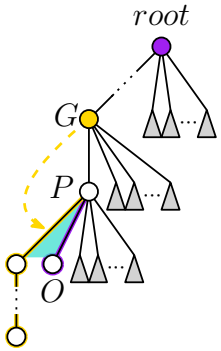


Current tree $\mathcal{T}$ where $P = root$ or O is not a leaf.

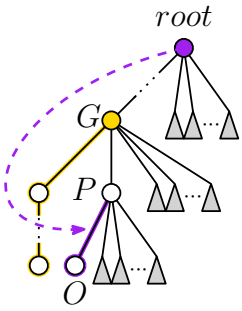


New tree $\mathcal{T}'$ is obtained by $\mathsf{pull}_{\mathcal{T}}(O, P)$.

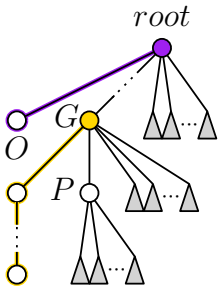
(a) Case (4.1a). After the pull, the node O is updated to be the new second-child of O (if non-null) or the new second-child of P (if non-null), or *null*. The pull is equivalent to $\mathsf{pushchild}(O, \mathsf{popchild}(P))$.



Current tree $\mathcal{T}$ where $P \neq root$ and O is a leaf.



Intermediate tree $\mathcal{I}$ is obtained by $\mathsf{pull}_{\mathcal{T}}(G, P)$.



New tree $\mathcal{T}'$ is obtained by $\mathsf{pull}_{\mathcal{I}}(root, P)$.

(b) Case (4.1b). After the pull, the node O is updated to be the new second-child of $root$. The pull operations are equivalent to $\mathsf{pushchild}(G, \mathsf{popchild}(P))$ followed by $\mathsf{pushchild}(root, \mathsf{popchild}(P))$

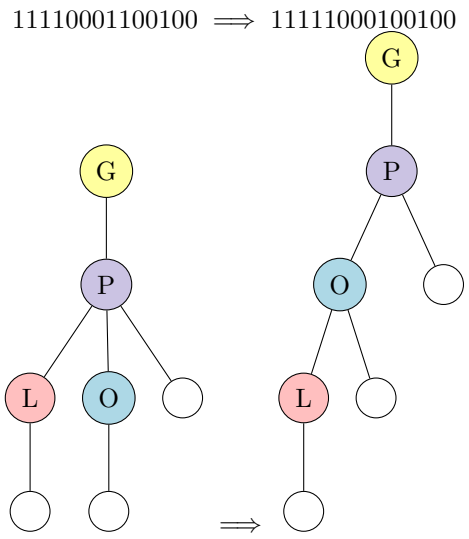
Figure 4.2: Our pull Gray code is generated by the two case successor rule in (4.1). This rule describes how to transform the current ordered tree $\mathcal{T}$ into the new next ordered tree $\mathcal{T}'$ using either one or two pulls. In these figures, O is the first node in a preorder traversal that is not on the path from the root to the leftmost descendent, and P is its parent, and G is its grandparent (if applicable). White circles denote non-null nodes, and grey triangles denote an unspecified number of children and subtrees. The pull operations, which are always path-pulls, are highlighted. The first pulling node and pulled path are in gold, while the second are in purple. The captions also explain how to update the value of O.

4.4 Proof of Correctness

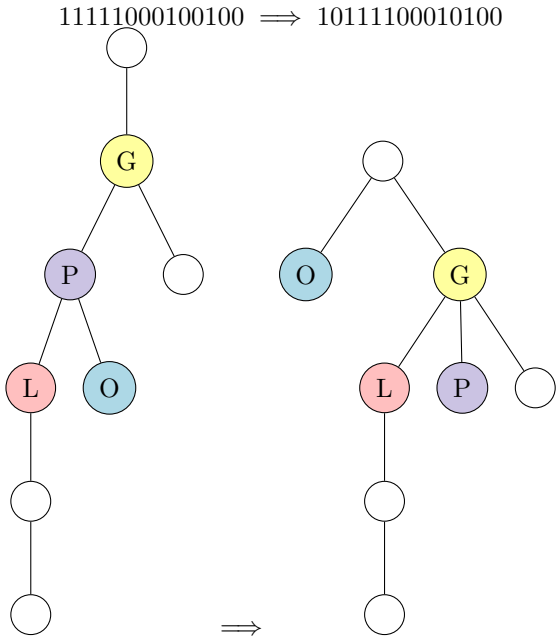
This section will prove that the successsor rule does in fact in (4.1) generates all ordered trees with n nodes. See [LW22b], section 5 for an alternate proof of this result, available here.

4.4.1 Definitions and Remarks

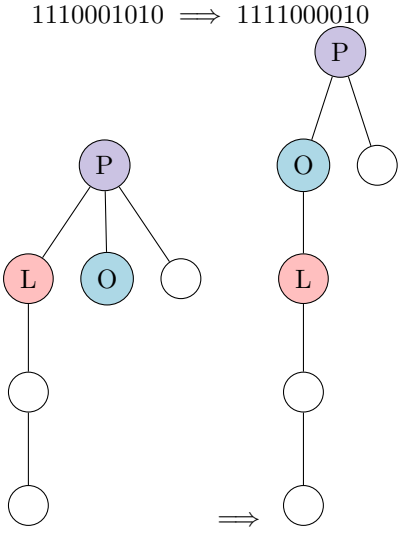
Let $\mathcal{T}$ be an ordered tree with $n+1$ nodes listed in preorder as $t_0, t_1, ..., t_n$. Note that t_0 is the root of $\mathcal{T}$. We define the *left-down path* of $\mathcal{T}$, $\mathsf{leftpath}(\mathcal{T})$, to be the unique path between $\mathcal{T}$'s root and $\mathcal{T}$'s leftost leaf. Let $D = \mathsf{Dyck}(T)$. Let x be the index of the first 1 following a 0 in D, mirroring section 3.3. Let s be the number of consecutive ones to start D and let z be the number of consecutive zeroes starting at d_{s+1}. Note that $z = (x - s - 1)$; $d_x = 1$. Let $\mathsf{oneindex}(D, i) =$ be the index of the $\underline{i^{th}}$ one in D. The following remarks can be derived from from the above definitions and the bijection between Dyck words and ordered trees given in given in 2.4.2.



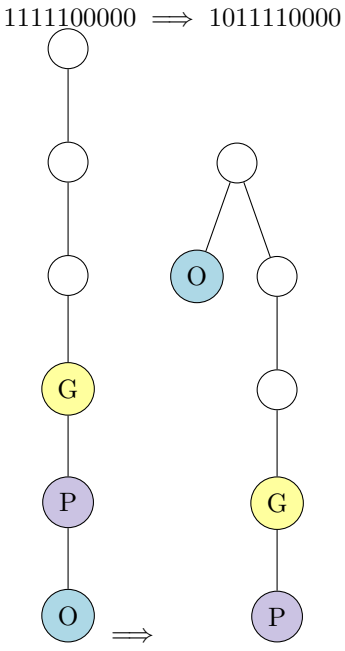
(a) O has at least 1 child: Case (4.1a)
$\mathsf{pull}(O, P)$
L becomes O's first child. The new first branching is between O and O's second child.



(b) $P \neq root$, O has no children: Case (4.1b)
$\mathsf{pull}(G, P); \mathsf{pull}(root, P)$
L becomes G's first child; O becomes the first child of $root$. The new first branching is between the root and the root's second child.



(c) O has no children, $P = root$: Case (4.1a)
$\mathsf{pull}(O, P)$
L becomes O's first child. The new first branching is between P and P's second child.



(d) $\mathcal{T}$ has no first branching: Case (4.2a)
$\mathsf{pull}(root, P)$
O becomes the first child of the root
The new first branching is between the root and the root's second child.

Figure 4.3: Illustrating the pull shifts on specific trees.

Remark 4.1. $\mathsf{Depth}(O) = s - z + 1$

Proof. t_s is the last node in the $\mathsf{leftpath}(\mathcal{T})$, as the left-down path has $s+1$ nodes starting at t_0. t_s has depth s, as it is exactly s steps from the root. Note that $O = t_{s+1}$. The number of zeroes between t_s and t_{s+1} is the number of zeroes between the $s^{\underline{th}}$ and $(s+1)^{\underline{st}}$ ones in D_i.

□

Remark 4.2. *O corresponds to D_x*

Proof. Recall that O is the first branching of $\mathcal{T}$ and the first second child visited in a preorder traversal of $\mathcal{T}$.

Recall that in the bijection between ordered trees and Dyck words given in 2.4.2 that each 1 in D corresponds to a step down and each 0 corresponds to a step up. A second child in a preorder traversal of an ordered tree must be equal to the

□

Remark 4.3. *Every non-leaf node below P in $\mathsf{leftpath}(\mathcal{T})$ has exactly 1 child.*

Proof. Suppose by way of contradiction that a node below P in $\mathsf{leftpath}(\mathcal{T})$ had a second child. That child would not be in $\mathsf{leftpath}(\mathcal{T})$ and would be be traversed before O in preorder. O was specified to be the first node in a preorder traversal of T that is not in $\mathsf{leftpath}(\mathcal{T})$, which generates a contradiction.

□

Remark 4.4. *L corresponds to D_{s-z+1}*

Proof. $\mathsf{Depth}(L) = \mathsf{Depth}(O) = s - z - 1$ since L and O are siblings. Therefore, L must be $s - z - 1$ steps down from the root $\implies$ L is the $s-j-1$th node in a preorder traversal of T $\implies$ T corresponds to D_{s-z-1}.

□

Remark 4.5. *The $i^{\underline{th}}$ non-root node in a preorder listing of $\mathcal{T}$ corresponds to the $i^{\underline{th}}$ one in D.*

Proof. Recall the method of constructing an ordered tree from a Dyck word. Each one in D creates a new node; zeroes in D do not create nodes. Generating an ordered tree from a Dyck word generates the nodes of the tree in preorder. Thus, t_i corresponds to the ith one in D for $1 \leq i \leq n$. □

Remark 4.6. *The difference in depths between nodes t_i and t_{i-1} is equal to one minus the number of zeroes between the $(i-1)^{\underline{st}}$ and $i^{\underline{th}}$ and ones in D*

Proof. This remark can be stated formally as

$$\mathsf{Depth}(t_i) - \mathsf{Depth}(t_{i-1}) = 1 - (\mathsf{oneindex}(D, i) - \mathsf{oneindex}(D, i-1) - 1) \tag{4.3}$$

Note that $(\mathsf{oneindex}(D, i) - \mathsf{oneindex}(D, i-1) - 1)$ is equal to the number of zeroes between the $i^{\underline{th}}$ and $(i-1)^{\underline{st}}$ ones in D.

This follows naturally from the bijection between Dyck words and ordered trees. Each zero corresponds to a step up in the tree before adding the next child.

If there are zero zeroes between the $i^{\underline{th}}$ and $(i-1)^{\underline{st}}$ ones in D, t_i is a child of t_{i-1}; $\mathsf{Depth}(t_i) = \mathsf{Depth}(t_{i-1}) + 1$

If there is one zero between the $i^{\underline{th}}$ and $(i-1)^{\underline{st}}$ ones in D, t_i is a child of t_{i-1}'s parent; $\mathsf{Depth}(t_i) = \mathsf{Depth}(t_{i-1}) + 1$.

Each subsequent zero between t_{i-1} and t_i decreases $\mathsf{Depth}(t_i)$ by one. Thus, the depth of t_i is the depth of t_{i-1} plus 1 minus the number of zeroes between t_{i-1} and t_i. □

Remark 4.7. *A preorder listing of $\mathsf{Depth}(t_i)$ for each $t_i \in T$ can be used to construct a Dyck word.*

Proof. Let $\mathcal{T} = t_0, t_1, ...t_n$ be a preorder traversal of T. Note that t_0 is the root of $\mathcal{T}$

Construct D as follows:

- Let $D = \epsilon$
- For each t_i, $1 \leq i \leq n$

- Append a 1 to D
- Append $1 - \mathsf{Depth}(t_i) + \mathsf{Depth}(t_{i-1})$ zeroes to D.

- Append $\mathsf{Depth}(t_n)$ zeroes to D.

□

4.4.2 Correspondence between coolCat and nextree

Recall that the successor rule $\overleftarrow{\mathsf{coolCat}}(D)$ generates all Dyck words. Therefore, to prove that $\mathsf{nextree}(\mathcal{T})$ generates all ordered trees with $|T|$ nodes, it is sufficient to show that, given an arbitrary ordered tree T and its corresponding Dyck word D, $\mathsf{nextree}(\mathcal{T})$ behaves the same on T as $\overleftarrow{\mathsf{coolCat}}(D)$ does on D. More precisely, we aim to prove the following:

Theorem 4.1. *Given an ordered tree T,* $\mathsf{nextree}(\mathcal{T}) = \mathsf{OTree}(\overleftarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)))$

Proof. $\overleftarrow{\mathsf{coolCat}}(D)$ and $\mathsf{nextree}(\mathcal{T})$ are each broken down into 3 cases in equations (3.8) and (4.1) respectively.

For convenience, equations (4.4) and (4.5) give the expanded restatemtents of the successor rules for $\mathsf{nextree}(\mathcal{T})$ and $\overleftarrow{\mathsf{coolCat}}(D)$ to facilitate comparisons between the two.

$$\mathsf{nextree}(\mathcal{T}) = \begin{cases} \mathsf{pull}(root, P) & \mathsf{leftpath}(\mathcal{T}) = \mathcal{T} & \text{(4.4a)} \\ \mathsf{pull}(O, P) & \text{if O has at least 1 child} & \text{(4.4b)} \\ \mathsf{pull}(G, P); \mathsf{pull}(root, P) & \text{if } P \neq root \text{ and O has no children} & \text{(4.4c)} \\ \mathsf{pull}(O, P) & \text{if O has no children and } P = root & \text{(4.4d)} \end{cases}$$

$$\overleftarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}_D(2n) & \text{if } D \text{ has no 01 substring} & \text{(4.5a)} \\ \mathsf{preshift}_D(x+1) & D_{x+1} = 1 & \text{(4.5b)} \\ \mathsf{preshift}_D(x+1) & D_{x+1} = 0 \text{ and } s > \frac{x-1}{2} & \text{(4.5c)} \\ \mathsf{preshift}_D(x) & D_{x+1} = 0 \text{ and } s = \frac{x-1}{2} & \text{(4.5d)} \end{cases}$$

We will show the following equivalences:

- (4.4a) corresponds to (4.5a)
- (4.4b) corresponds to (4.5b)
- (4.4c) corresponds to (4.5c)
- (4.4d) corresponds to (4.5d)

To accomplish this, we will first prove a few auxillary lemmas to be used to show equivalency between cases.

Let $D = \mathsf{Dyck}(T)$, s be the number of consecutive ones to start D, and z be the number of consecutive zeroes starting at d_{s+1}. Note that $z = (x - s - 1)$; $d_x = 1$

Lemma 4.2. *D has no 01 substring* $\iff$ $\mathsf{leftpath}(\mathcal{T}) = \mathcal{T}$

Proof. If D has no 01 substring, $D = 1^n 0^n$, and T is $n+1$ nodes where t_0 is the root and each t_i for $1 \leq i \leq n$ is a child of t_{i-1} In this case, $\mathcal{T}$ is a single path of $n+1$ nodes, and the left-down path of T is the entire tree. □

Lemma 4.3. $D_{x+1} = 0 \iff$ *O has no children*

Proof. This follows logically from the bijection between Dyck words and ordered trees. D_x corresponds to O. If $D_{x+1} = 0$, an "upward" step is taken after O and consequently the next node after O cannot be a child of O. Since the ones in D give the nodes of T in preorder, O must have no children.

Informally, once you go "up" from O, the bijection between Dyck words and ordered trees gives no way to go "back down" to give O an additional child. □

Lemma 4.4. $P = root \iff s = z = \frac{x-1}{2}$.

Proof. First, note that $P = root$ simply means that O is a child of the root. O is a child of the root $\iff \mathsf{Depth}(O) = 1$. Additionally, note that $s + z = x - 1$

As shown in remark 4.1, $\mathsf{Depth}(O) = s - z + 1$. Therefore, $P = root \iff s = z = \frac{x-1}{2}$ i.e. the first $x - 1$ symbols of D are $\frac{x-1}{2}$ ones followed by $\frac{x-1}{2}$ zeroes.

□

Lemma 4.5. (4.4a) *corresponds to* (4.5a)

Proof. Let $D = \mathsf{Dyck}(T)$

Per lemma 4.2 D has no 01 substring $\iff \mathsf{leftpath}(\mathcal{T}) = \mathcal{T}$.

Thus, $\mathsf{nextree}(\mathcal{T})$ executes case (4.4a) if and only if $\overleftarrow{\mathsf{coolCat}}(D)$ executes case (4.5a)

Note that since D has no 01 substring, $D = 1^n0^n$.

Additionally, since $\mathsf{leftpath}(\mathcal{T}) = \mathcal{T}$, T can be specified as follows.

$\mathcal{T} =$

node	t_0	t_1	t_2	...	t_{n-1}	$F = t_s = t_n$
depth	0	1	2	...	$n-1$	n
Dyck		1^n0^n				

The third row of this table illustrates the construction of $\mathsf{Dyck}(T)$ via the process specified in remark 4.7.

Shifting F to be the first child of the root changes $\mathsf{Depth}(F)$ to 1 and does not affect the depth of any other nodes. Thus, if $\mathcal{T}' = \mathsf{nextree}(\mathcal{T})$,

$\mathcal{T} =$

node	t_0	$F = t_s = t_n$	t_1	t_2	...	t_{n-1}
depth	0	1	1	2	...	$n-1$
Dyck		1	$01^{n-1}0^{n-1}$			

Recall that $\overleftarrow{\mathsf{coolCat}}(D) = \mathsf{preshift}_D(2n)$ if D has no 01 substring. $D_{2n} = 0$, and therefore $\overleftarrow{\mathsf{coolCat}}(D) = 101^{n-1}0^{n-1}$

Note that this is exactly the Dyck word constructed from $\mathcal{T}'$. Therefore, if D has no 01 substring or $\mathsf{leftpath}(\mathcal{T}) = \mathcal{T}$,

$\mathsf{OTree}(\overleftarrow{\mathsf{coolCat}}(D)) = \mathsf{nextree}(\mathcal{T})$

□

Lemma 4.6. (4.4c) *corresponds to* (4.5c)

Proof. Let $D = \mathsf{Dyck}(T)$

Per lemma 4.4 $P = root \iff D$ starts with exactly $\frac{x-1}{2}$ ones.

It was also previously shown that $D_{x+1} = 0 \iff O$ has no children. Thus, $\mathsf{nextree}(\mathcal{T})$ executes case (4.1b) if and only if $\overleftarrow{\mathsf{coolCat}}(D)$ executes case (4.5c)

We now show that the execution of (4.1b) is equivalent to the execution of (4.5c) given case a. Given $\mathsf{Dyck}(T) = D = 1^s0^z10d_{x+2}d_{x+3}...d_{2n}$, we aim to show that

$\mathsf{Dyck}(\mathsf{nextree}(\mathcal{T})) = \overleftarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T))$

Note that in this case $\mathsf{nextree}(\mathcal{T})$ can be obtained by performing $\mathsf{pull}(G, P); \mathsf{pull}(root, P)$.

Let $\mathcal{T}' = \mathsf{pull}_T(P, G)$; $\mathcal{T}'' = \mathsf{pull}_{T'}(P, root)$

Note that $\mathsf{nextree}(\mathcal{T}) = \mathcal{T}''$

Since $P \neq root$, we know that G, the parent of P, exists. Thus, we can assume that $G, P, L \in \mathsf{leftpath}(\mathcal{T})$. T can therefore be specified as follows:

$\mathcal{T} =$

node	t_0	t_1	...	$G = t_{s-z-1}$	$P = t_{s-z}$	$L = t_{s-z+1}$	...	$F = t_s$	$O = t_{s+1}$	...
depth	0	1	...	$(s-z-1)$	$(s-z)$	$(s-z+1)$	...	s	$(s-z+1)$	...
Dyck		1^s							0^z1	$0\ldots$

Furthermore, recall that L (and all other non-leaf nodes $\in \mathsf{leftpath}(\mathcal{T})$ must have exactly one child. Therefore, every node below L in $\mathsf{leftpath}(\mathcal{T})$ has its depth reduced by one; no other nodes have their depth affected by this shift. Therefore, T' can be written as follows:

$\mathcal{T}' =$

node	t_0	t_1	...	$G = t_{s-z-1}$	$L = t_{s-z+1}$	...	$F = t_s$	$P = t_{s-z}$	$O = t_{s+1}$	...
depth	0	1	...	$(s-z-1)$	$(s-z)$	...	$s-1$	$(s-z)$	$(s-z+1)$	...
Dyck		1^{s-1}						0^z1	1	$0\ldots$

Since L is now G's first child, P changes from being G's first child to G's second child. P is therefore removed from the left-down path of $\mathcal{T}'$, thereby making P the first node in a preorder traversal of $\mathcal{T}'$ that is not in the left-down path of $\mathcal{T}'$. Therefore, $|\mathsf{leftpath}(\mathcal{T}')| = s$; $O' = P$.

Recovering a Dyck word from $\mathcal{T}'$, we obtain

D'=$1^{s-1}0^z110d_{x+2}d_{x+3}, \ldots, d_{2n}$

Next, we use $\mathsf{pull}_{T'}(P, root)$ to obtain $\mathcal{T}'' = \mathsf{nextree}(\mathcal{T})$

$\mathsf{pull}_{T'}(P, root)$ shifts O to become the first child of the root. Note that we know that O has no children. Consequently, no nodes other than O have their depth affected by this shift. Thus,

$\mathcal{T}'' =$

node	t_0	$O = t_{s+1}$	t_1	t_2	...	$G = t_{s-z-1}$	$L = t_{s-z+1}$	...	$F = t_s$	$P = t_{s-z}$	...
depth	0	1	1	2	...	$(s-z-1)$	$(s-z)$	...	$s-1$	$(s-z)$	...
Dyck		1	01^{s-1}							0^z1	...

Therefore, since $\mathcal{T}'' = \mathsf{nextree}(\mathcal{T})$, $\mathsf{Dyck}(\mathsf{nextree}(\mathcal{T})) = 101^{s-1}0^z1\ldots$

Since $\mathsf{Dyck}(T) = D = 1^s0^z10\ldots$ (4.5b) gives that

$\overleftarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)) = 101^{s-1}0^z1\ldots$

Therefore, we have shown that $\mathsf{Dyck}(\mathsf{nextree}(\mathcal{T})) = \overleftarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)) = 101^{s-1}0^z1\ldots$

□

Lemma 4.7. (4.4b) *corresponds to* (4.5b)

Proof. Per 4.3, as O has at least 1 child $\iff$ $D_{x+1} = 1$.

Thus, $\mathsf{nextree}(\mathcal{T})$ will execute case (4.4b) if and only if $\overleftarrow{\mathsf{coolCat}}(D)$ executes case (4.5b)

Therefore, we aim to show that, given O has at least one child and $D_{x+1} = 1$,

$\mathsf{preshift}_{\mathsf{Dyck}(T)}(x+1) = \mathsf{Dyck}(\mathsf{pull}_T(P, O))$

Since $D_{x+1} = 1$, we can rewrite D as. $D = 1^s0^z11$

$\mathcal{T} =$

node	t_0	t_1	...	$G = t_{s-z-1}$	$P = t_{s-z}$	$L = t_{s-z+1}$	...	$F = t_s$	$O = t_{s+1}$	$t_{s+2}\ldots$
depth	0	1	...	$(s-z-1)$	$(s-z)$	$(s-z+1)$	...	s	$(s-z+1)$	$s-z+2\ldots$
Dyck		1^s							0^z1	$1\ldots$

$\mathsf{pull}_T(P, O)$ shifts L to be O's first child:

Nodes $L = t_{s-z+1}$ through $F = t_s$ will now come after O in preorder traversal. Additionally, $\mathsf{leftpath}(\mathcal{T})$ will now go through O; every node in $\mathsf{path}(\mathsf{T}, \mathsf{L}, \mathsf{F})$ will have its depth increased by one.

Therefore, $\mathcal{T}' = \mathsf{nextree}(\mathcal{T})$ can be specified as follows:

$\mathcal{T}' =$

node	t_0	t_1	$\dots$	$G = t_{s-z-1}$	$P = t_{s-z}$	$O = t_{s+1}$	$L = t_{s-z+1}$	$\dots$	$F = t_s$	$t_{s+2}\dots$
depth	0	1	$\dots$	$(s-z-1)$	$(s-z)$	$(s-z+1)$	$(s-z+2)$	$\dots$	$s+1$	$s-z+2\dots$
Dyck		1^{s+1}								$0^z1\dots$

Note that $z \geq 1$, so z zeroes occur between the one corresponding to t_s and the one corresponding to t_{s+2}.

Next, recall that $D = \mathsf{Dyck}(T) = D = 1^s0^z11\dots$ and that $x = s + z + 1$

Therefore, $\overleftarrow{\mathsf{coolCat}}(D) = \mathsf{preshift}_D(x+1) = 1^{s+1}0^z1\dots$, which is the same as the Dyck word resulting from translating $\mathcal{T}' = \mathsf{nextree}(\mathcal{T})$ to the Dyck word $1^{s+1}0^z1\dots$

□

Lemma 4.8. (4.4d) *corresponds to* (4.5d)

Proof. $\mathcal{T} \neq \mathsf{leftpath}(\mathcal{T}) \iff D$ has a 01 substring.

$D_{x+1} = 1 \iff O$ has at least one child.

$D_{x+1} = 0$ and $s = \frac{x-1}{2} \iff O$ has no children and O is a child of the root.

O has no children and P=root. Therefore $s = z$, $x = 2s + 1$

We can thus rewrite $D = \mathsf{Dyck}(T) = 1^s0^s101\dots$

Furthermore, since $s = z$, O has depth 1.

Therefore, we can write T as $\mathcal{T} =$

node	$P = t_0$	$L = t_1$	$\dots$	$F = t_s$	$O = t_{s+1}$	$t_{s+2}\dots$
depth	0	1	$\dots$	s	1	1
Dyck		1^s			0^s1	$01\dots$

$\mathsf{pull}_T(P, O)$ shifts L to be O's first child:

Therefore, nodes $L = t_1$ through $F = t_s$ will now come after O in preorder traversal. Additionally, $\mathsf{leftpath}(\mathcal{T})$ will now go through O; every node in $\mathsf{path}(\mathsf{T}, \mathsf{L}, \mathsf{F})$ will have its depth increased by one.

Therefore, $\mathcal{T}' = \mathsf{nextree}(\mathcal{T})$ can be specified as follows:

$\mathcal{T}' =$

node	$P = t_0$	$O = t_{s+1}$	$L = t_1$	$\dots$	$F = t_s$	$t_{s+2}\dots$
depth	0	1	2	$\dots$	$s+1$	1
Dyck		1^{s+1}				$0^{s+1}1\dots$

Since $D = \mathsf{Dyck}(T) = 1^s0^s101\dots$, $\overleftarrow{\mathsf{coolCat}}(D) = 1^{s+1}0^{s+1}1\dots$ as per case (4.4d). This is identical to the Dyck word constructed from $\mathcal{T}' = \mathsf{nextree}(\mathcal{T})$. Therefore, cases (4.4d) and (4.5d) are equivalent.

□

Since these 4 cases cover all cases for the two successor rules, we have shown that $\mathsf{nextree}(\mathcal{T}) = \mathsf{OTree}(\overleftarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)))$ in all cases. □

Chapter 5

Loopless Ordered Tree Generation

This chapter provides two loopless implementations of the Gray code given in Chapter 4. Section 5.1 discusses this chapter's algorithm in the context of other related results. Section 5.2 provides an implementation using a linked-list like structure for storing children. Secion 5.3 provides an implementation using an array to store children.

The algorithms for both the linked-list and array representations of ordered trees take the following approach. They begin by creating an initial tree $\mathcal{T}_1$ with $n+1$ nodes such that $\mathsf{Dyck}(\mathcal{T}_1) = 101^{n-1}0^{n-1}$. This tree can be generated by creating a path with $n+1$ nodes and executing $\mathsf{pull}(root, P)$ where P is the parent of the path's leaf. Both algorithms then repeatedly apply the successor rule from equation 4.1. The algorithms keep track of O using the observation that if O has at least one child before shifting, the new first branching is between O and O's second child after shifting. If O has no children before shifting, the new first branching is between P and P's second child after shifting. Figures 5.2 and 5.4 provide pseudocode and a C implementation for this algorithm using each tree representation. Full source code for both of these algorithms in C is available here.

5.1 Relationship to Previous Results

The algorithms in sections 5.2 and 5.3 both generate ordered trees *looplessly*, meaning that each tree is generated in worst-case constant time. This is faster than other algorithms for generating ordered trees which take constant amortized time [PM21] [Er85] [Zak80] [Ska88].

Taken in conjunction with Ruskey and Williams's algorithms for Dyck words and binary trees, this algorithm completes a trio of loopless cool-lex algorithms for enumerating the three foremost Catalan structures. Additionally, like the cool-lex algorithm for binary trees, this algorithm generates ordered trees stored as pointer structures. This contrasts from other efficient Gray codes for enumerating ordered trees, which use either bit-strings or integer sequences to represent ordered trees [PM21] [Zak80] [Er85] as representations of ordered trees. Skarbek's 1988 paper *Generating Ordered Trees* gives a constant amortized time algorithm for generating ordered trees stored as pointer structures and is therefore a notable exception to this [Ska88]. Generating ordered trees via a pointer structure facilitates the practical use of the trees generated by this algorithm, as a translation step between an alternative representation and a tree structure is not necessary to traverse the tree .

Korsh and Lafollette [KL00] gave a loopless algorithm for generating ordered trees with a fixed branching sequence. In other words, they generated subsets of ordered trees with a fixed nuimber of nodes for which a listing of the number of children at each node form the same nultiset. Their algorithm used a string-based representation rather than a link-based representation, and thus answered a challenging open problem posed by van Baronaigen [VB91]. They also stated, in one sentence, that their results could be adapted to a link-based representation. By 'layering' the output of that proposed algorithm, it may be possible to create a loopless link-based algorithm for generating all ordered trees with n nodes. It is also important to note that the results in [KL00] are not simple: the provided C code includes over one hundred instance of the `if` and `else` keywords.

```
typedef struct node {
  struct node* parent;
  struct node* first;
  struct node* right;
} node;

void pull(node *A, node *B){
  // A pulls B's first child
  node *pulled = B->first;
  B->first = pulled->right;
  pulled->right = A->first;
  A->first = pulled;
  pulled->parent = A;
}

node* kthchild(node* parent, int k){
  node* curr=parent->first;
  for(int i = 1; i < k; i++){
    curr=curr->right;
  }
  return curr;
}
```

Figure 5.1: Struct definitions and implementations for $\mathsf{pull}(A, B)$ and accessing a $k^{\underline{th}}$ child for an link-based ordered tree structure.

5.2 Storing Children in a Linked List

The implementation in this section uses a linked structure to store a node's children, where an ordered tree node stores pointers to its first (leftmost) child and its right sibling. Therefore, to access a node's 3rd child, one would use `node->first->right->right`. This has the advantages of space efficiency and $O(1)$ appending and prepending to a node's list of children, but the disadvantage of requiring $O(k)$ time to access a node's $k^{\underline{th}}$ child.

Figure 5.1 provides the C struct definition for this ordered tree definition as well as an implementation of $\mathsf{pull}(A, B)$ and a helper function for accessing a node's $k^{\underline{th}}$ child.

The algorithm in Figure 5.2 uses a while loop with two if statements per successor to evaulate the cases in (4.1). The while loop evaluates if there is a first branching: if `NULL` is assigned to O, then there is no first branching, and the order is complete. The first if statement evaluates whether O has at least 1 child: O's first child is not `NULL` if and only if O has at least 1 child. The second if statement evaluates whether P is the root of the tree. The algorithm executes $\mathsf{pull}(O, P)$ if O has at least child or P is the root of the tree. Otherwise, it executes $\mathsf{pull}(G, P)$ and $\mathsf{pull}(root, P)$. The algorithm keeps track of O via the observation that if O has at least one child before shifting, the new first branching is between O and O's second child after shifting and if O has no children before shifting, the new first branching is between P and P's second child after shifting. This translates to assigning `o = o->first->right` if O has at least 1 child and `o = o->right` otherwise.

This implementation is very clearly loopless, as $\mathsf{pull}(A, B)$ is a constant time operation and only two if statements and the condition of the while loop are evaluated per successor.

Generate all ordered trees with $t+1$ nodes

```
function COOL-ORDERED-TREES(t)
    ▷ Generate initial tree
    O ← root.first
    visit(root)
    while O ≠ NULL do
        P ← O.parent
        if O.first ≠ NULL then
            pull(O, P)
            O ← O.first.right
        else
            if P == root then
                pull(O, P)
            else
                pull(O, P)
                pull(root, P)
            O ← O.right
        visit(root)
```

```
void coolOtree(int n) {
  node* root = get_initial_tree(n);
  node* o=root->first->right;
  visit(root);
  while (o) {
    p=o->parent;
    if (o->first) {
      pull(o,p);
      o = o->first->right;
    } else {
      if (p == root) {
        pull(o,p);
      } else {
        pull(p->parent,p);
        pull(root,p);
      }
      o = o->right;
    }
    visit(root);
  }
}
```

Figure 5.2: Pseudocode and C implementation for linked-list tree representation.

5.3 Storing Children in an Array

The second implementation uses an array-like approach to storing children. This has the advantage of $O(1)$ access time for all children but the disadvantage of either requiring $O(n)$ space for each node or the additional cost of resizing arrays. Each node also stores a counter for its number of children and a number keeping track of the maximum amount of children it can store (before requiring reallocation)). This implementation stores the list of children "backwards," so that the array can be operated on like a stack. In particular, this allows for pushing and popping first children without shifting the whole array. Each `node->children[0]` is set to and kept as `NULL` for a null-termination like effect that is useful in the algorithm. Therefore, to access a node's $k^{\underline{th}}$ child, one would use `node->children[(node->nch)-k+1]`. The `kthchild` function simplifies accessing a node's $k^{\underline{th}}$ child and is used in our implementation. Figure 5.3 provides the C struct definition for this ordered tree definition as well as an implementation of pull(A, B) and a `kthchild`.

Like the linked-list implementation in Figure 5.2, the array based implementation in Figure 5.4 uses a while loop with two if statements per successor to evaulate the cases in (4.1). The logical structure of the while loop and if statements is identical to that of 5.4: the while loop terminates when there is no first branching, the first if statement evaluates whether O has at least 1 child, and the second if statement evaluates whether P is the root of the tree. The algorithms in 5.2 and 5.4 are almost identical apart from the structure they use and the implementation differences between between `pull` and `apull`.

```
typedef struct arraynode {
    int nch;
    int maxch;
    struct arraynode* parent;
    struct arraynode** children;
} anode;

void apull(anode* A, anode* B){
    //A pulls B's first child
    anode* pulled = B->children[B->nch--];
    A->children[++(A->nch)]=pulled;
    pulled->parent=A;
}

anode* kthchild(anode* parent, int k){
  return parent->children[parent->nch-k+1];
}
```

Figure 5.3: Struct definitions and implementations for $\mathsf{pull}(A, B)$ and accessing a $k^{\underline{th}}$ child for an array-based ordered tree structure.

Generate all ordered trees with $n + 1$ nodes

```
function COOL-ORDERED-TREES(n)
    ▷ Generate initial tree
    O ← kthchild(root, 2)
    visit(root)
    while O ≠ NULL do
        P ← O.parent
        if O.nch > 0 then
            pull(O, P)
            O ← kthchild(O, 2)
        else
            if P == root then
                pull(O, P)
            else
                pull(P.parent, P,)
                pull(root, P,)
            O ← kthchild(O.parent, 2)
        visit(root)
```

```
void coolAOtree(int t, void (*visit)(anode*)){
  anode* root = get_initial_atree(t);
  anode *p, *o=kthchild(root,2);
  visit(root);
  while(o){
    p = o->parent;
    if(o->nch){
      apull(o,p);
      o=kthchild(o,2);
    }else{
      if(p == root){
        apull(o,p);
      }else{
        apull(p->parent,p);
        apull(root,p);
      }
      o=kthchild(o->parent,2);
    }
    visit(root);
  }
}
```

Figure 5.4: Pseudocode and C implementation for array-based tree representation.

Chapter 6

Lattice Paths: Łukasiewicz, Motzkin, Schröder

This chapter discusses Motzkin, Schröder, and Łukasiewicz paths, combinatorial objects that will be generated by the algorithm in Chapter 7. Motzkin, Schröder, and Łukasiewicz paths all provide generalizations of Dyck paths. All three languages retain the requirement that the path start at the origin, end on the x axis, and never step below the x axis. However, the three languages vary from Dyck paths in the types of steps they allow. Section 6.1 will discuss Motzkin paths, Section 6.2 Schröder paths, and Section 6.3 will discuss Łukasiewicz paths.

6.1 Motzkin Paths

Recall the interpretation of Dyck words as paths in the Cartesian plane from Section 2.1. Motzkin paths use (1,0) horizontal steps in addition to (1,1) and (1,-1) steps. The number of Motzkin paths termminating at $(n, 0)$ is counted by the $n^{\underline{th}}$ Motzkin number. This sequence is illustrated below for $n \geq 0$ along with its corresponding OEIS entry.

$$\mathcal{M}_n = 1, 1, 2, 4, 9, 21, 51, \ldots \qquad \text{OEISA001006} \tag{6.1}$$

(6.2)

The Motzkin numbers are named for $20^{\underline{th}}$ century Israeli-American mathematician Theodore Motzkin and are closely related to the Catalan numbers. In particular, the Motzkin numbers can be expressed in terms of binomial coefficients and Catalan numbers via the following equation:

$$\mathcal{M}_n = \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{2k} \mathcal{C}_k \tag{6.3}$$

And inversely,

$$\mathcal{C}_{n+1} = \sum_{k=0}^{n} \binom{n}{k} \mathcal{M}_k \tag{6.4}$$

The Motzkin numbers have many additional combinatorial interpretations, which are often similar to combinatorial interpretations of the Catalan numbers. For example, where $\mathcal{C}_n$ counts the number of ordered trees with $n+1$ nodes, $\mathcal{M}_n$ counts the number of ordered trees with $n+2$ nodes with the restriction that no node except for the root has exactly one child (although the root need not have exactly one child). Donaghey and Shapiro's *Motzkin Numbers* demonstrates this as well as many other interesting bijections [DS77].

6.2 Schröder Paths

Schröder paths are identical to Motzkin paths except they allow for $(2, 0)$ horizontal steps instead of $(1, 0)$. The number of Schröder paths terminating at $(2n, 0)$ is counted by the $n^{\underline{th}}$ big Schröder number. This sequence is illustrated below for $n \geq 0$ along with its corresponding OEIS entry.

$$\mathcal{S}_n = 1, 2, 6, 22, 90, 394, 1806, \ldots \qquad \text{OEISA006318} \tag{6.5}$$

The use of $(2, 0)$ horizontal steps seems arbitrary at first but has a number of useful properties. For example, unlike Motzkin paths, Schröder paths retain the property of Dyck paths that each path must terminate at $(2n, 0)$ for some integer n. Conversely, one-length horizontal step in Motzkin words allows Motzkin paths to termminate at odd positions on the x axis.

The big Schröder numbers are named for $19^{\underline{th}}$ century German mathematician Ernst Schröder. They are related to but distinct from the little Schröder numbers, which have the fascinating history of potentially being discovered thousands of years before the Catalan, Motzkin, and big Schröder numbers by the Greek astronomer Hipparchus. See [Sta97] for a brief history of this discovery and [Bob11] for an in-depth reconstruction of Hipparchus's logic.

Another combinatorial interpretation of the big Schröder numbers is the number of rectangular partitions constrained to lie on $n+1$ points [ABP04]. Shapiro and Sulanke's *Bijections for the Schröder Numbers* gives many other related interpretations of the Schröder numbers [SS00].

We encode both Motzkin and Schröder paths with southeast steps encoded as zeroes, horizontal steps as ones, and northeast steps as twos. Consequently, in the context of fixed-content generation, Motzkin and Schröder paths are identical. However, their Cartesian plane representations will differ in the length of horizontal steps. As a basic example, the Motzkin/Schröder word $2, 1, 1, 0$ terminates at the point $(4, 0)$ if interpreted as a Motzkin path and $(6, 0)$ if interpreted as a Schröder path. We refer to these integer based encodings as *Motzkin words* or *Schröder words*. Motzkin and Schröder words have the property that each prefix of the word has a sum at least as large as its length, and the whole word has a sum equal to its length.

6.3 Łukasiewicz Paths

Łukasiewicz paths allow $(1, -1)$ steps, $(1, 0)$ steps and any $(1, k)$ step where k is a positive integer. We encode each $(1, k)$ step in a Łukasiewicz path as $k + 1$ and therefore encode an Łukasiewicz path of length n as a sequence of integers. We call the resulting string of integers a *Łukasiewicz word*. Like Motzkin and Schröder words, Łukasiewicz words have the property that the sum of each prefix of a Łukasiewicz word is at least as great as the prefix's length and the sum of an entire Łukasiewicz word is equal to its length..

The number of Łukasiewicz paths terminating at $(n, 0)$ is counted by the $n^{\underline{th}}$ Catalan number, $\mathcal{C}_n$. Łukasiewicz paths are therefore in bijective correspondence with the Catalan objects discussed in in Chapter 2, sharing a particularly nice correspondence with ordered trees. In particular, a Łukasiewicz word of length n can be constructed from an ordered tree with $n + 1$ nodes via a preorder listing each ordered tree node's number of children, excluding the final leaf. See Figure 6.2 for an illustration of this bijection. Our algorithm in Chapter 7 will generate Łukasiewicz words with *fixed content*.

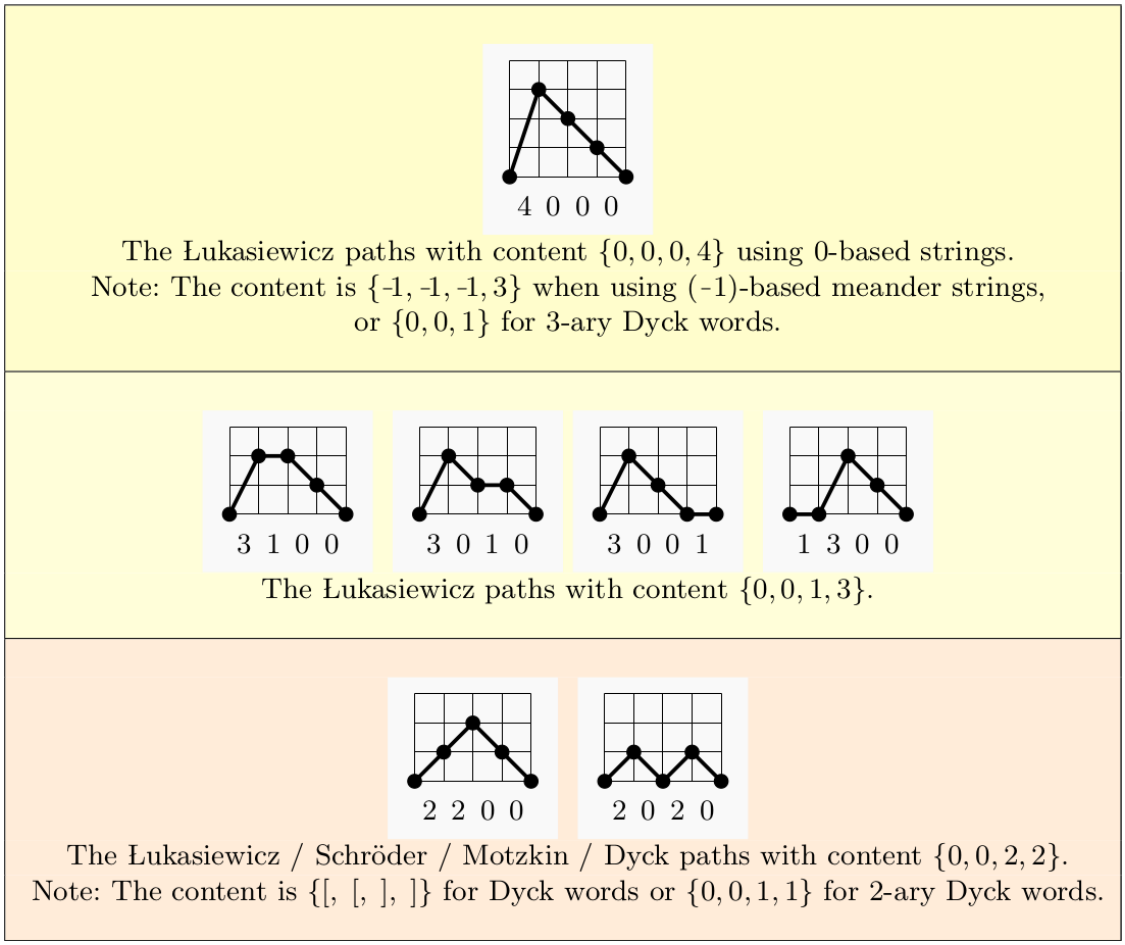


Figure 6.1: Łukasiewicz, Motzkin, and Schröder words.

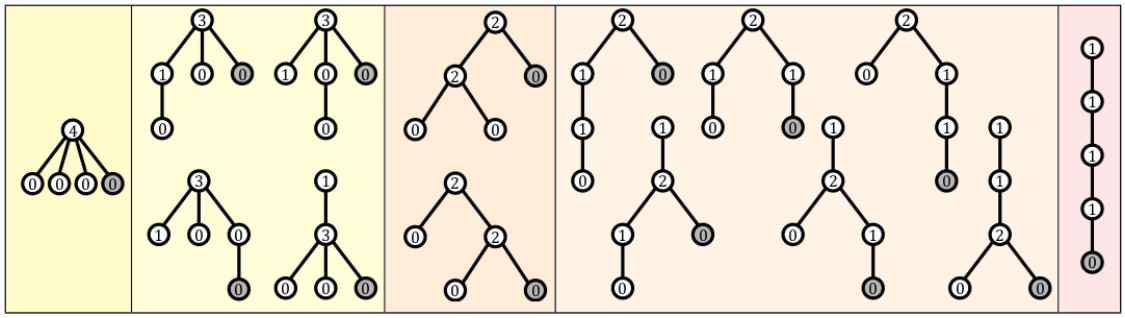


Figure 6.2: The $\mathcal{C}_4$=14 Łukasiewicz paths of order $n = 4$ are in bijective correspondence with the 14 rooted ordered trees with $n + 1 = 5$ nodes. Ordered tree nodes are labelled by their number of children. Given a tree, a Łukasiewicz word is obtained by recording the number of children of each node in preorder traversal; the zero from the rightmost leaf is omitted. For example, the two trees in the middle section correspond to 2200 (top) and 2020 (bottom) respectively.

Chapter 7

A New Shift Gray Code for Łukasiewicz Words

In this chapter, we give a Gray code for generating Łukasiewicz words with fixed content using left-shifts. Section 7.1 will give the successor rule for the Gray code; Section 7.2 will prove its correctness.

7.1 Successor Rule

In this section, we provide a successor rule that applies a left-shift to a Łukasiewicz word. The rule is given below in (7.1) Let S be a multiset whose sum is equal to its length. Let $\mathcal{L}(S)$ denote the set of valid Łukasiewicz words with content equal to S. Let $\alpha \in \mathcal{L}(S)$.

Recall the definition of a left-shift from equation (3.1): $\text{left}_\alpha(i, j)$ shifts the $j^{\underline{th}}$ symbol in α into the $i^{\underline{th}}$ position. We also define $\rho = \alpha_1, \alpha_2, \cdots \alpha_m$ to be the non-increasing prefix of α, i.e. the longest prefix of α such that $\alpha_i \geq \alpha_{i-1}$ for all $i \leq m$.

$$\overleftarrow{\text{luka}}(\alpha) = \begin{cases} \text{left}(n, 2) & \text{if } m = n & \text{(7.1a)} \\ \text{left}(m+1, 1) & \text{if } m = n-1 \text{ or } \alpha_m < \alpha_{m+2} \text{ or} & \text{(7.1b)} \\ & (\alpha_{m+2} = 0 \text{ and } \sum \rho = m) & \\ \text{left}(m+2, 1) & \text{if } \alpha_{m+2} \neq 0 & \text{(7.1c)} \\ \text{left}(m+2, 2) & \text{otherwise} & \text{(7.1d)} \end{cases}$$

Figure 7.1 illustrates the successor rule on every string in $\mathcal{L}(S)$ for $S = \{0, 0, 0, 1, 2, 3\}$. For example, consider the top row with $\alpha = a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 = 302100$. Here the non-increasing prefix is $a_1 \cdot a_2 = 30$, so $x = 1$, and the length of the string is $n = 6$. Thus, $x \neq n$, so (7.1a) is not applied. Now consider the conditions in (7.1b). The second condition is $a_m < a_{m+2}$, which is $a_2 = 0 < 1 = a_4$ for α. Since this is true, $\overleftarrow{\text{luka}}(\alpha) = \text{left}(m+1, 1)$ by (7.1b), which is $\text{left}(3, 1)$ for α. In other words, the rule left-shifts a_3 into position 1. Thus, the next string in the list is $a_3 \cdot a_1 \cdot a_2 \cdot a_4 \cdot a_5 \cdot a_6 = 230100$, as seen in the second row of Figure 7.1.

7.1.1 Observations

Note that (7.1) left-shifts a symbol that is at most two symbols past the non-increasing prefix. Thus, the shifts given by (7.1) are usually short, and the symbols at the right side of the string are rarely changed. This implies that the order will have some similarity to co-lexicographic order, which orders strings right-to-left by increasing symbols. In fact, the order turns out to be a cool-lex order, as discussed in Section 7.2.

7.2 Proof of Correctness

This section proves that the successor rule in (7.1) generates all Łukasiewicz words with a given set of content. Our strategy is to define a recursive order of $\mathcal{L}(S)$, and show that (7.1) creates the next

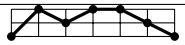
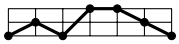
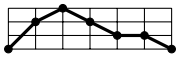
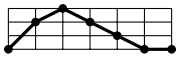
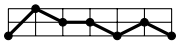
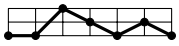
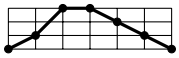
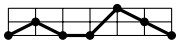
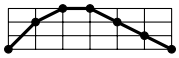
Łukasiewicz path	Łukasiewicz word	m	(7.1)	shift	scut
	302100	2	(7.1b)	left$(3,1)$	100
	230100	1	(7.1d)	left$(3,2)$	100
	203100	2	(7.1b)	left$(3,1)$	100
	320100	3	(7.1d)	left$(5,2)$	100
	302010	2	(7.1d)	left$(4,2)$	10
	300210	3	(7.1b)	left$(4,1)$	10
	230010	1	(7.1d)	left$(3,2)$	10
	203010	2	(7.1b)	left$(3,1)$	10
	320010	4	(7.1d)	left$(6,2)$	10
	302001	2	(7.1d)	left$(4,2)$	1
	300201	3	(7.1b)	left$(4,1)$	1
	230001	1	(7.1d)	left$(3,2)$	1
	203001	2	(7.1b)	left$(3,1)$	1
	320001	5	(7.1b)	left$(6,1)$	1
	132000	1	(7.1b)	left$(2,1)$	2000
	312000	2	(7.1d)	left$(4,2)$	2000
	301200	2	(7.1b)	left$(3,1)$	200
	130200	1	(7.1b)	left$(2,1)$	200
	310200	3	(7.1d)	left$(5,2)$	200
	301020	2	(7.1d)	left$(4,2)$	20
	300120	3	(7.1b)	left$(4,1)$	20
	130020	1	(7.1b)	left$(2,1)$	20
	310020	4	(7.1b)	left$(5,1)$	20
	231000	1	(7.1c)	left$(3,1)$	31000
	123000	1	(7.1b)	left$(2,1)$	3000
	213000	2	(7.1d)	left$(4,2)$	3000
	201300	2	(7.1b)	left$(3,1)$	300
	120300	1	(7.1b)	left$(2,1)$	300
	210300	3	(7.1b)	left$(4,1)$	300
	321000	6	(7.1a)	left$(6,2)$	ϵ

Figure 7.1: The left-shift Gray code cool(S) for Łukasiewicz words with content $S = \{0, 0, 0, 1, 2, 3\}$. Each row gives the non-increasing prefix length m, the rule (7.1), and the shift that creates the next word. The right column gives the scut of each string, which illustrates the suffix-based recursive definition of cool-lex order.

string in this order.

7.2.1 Terminology and Remarks

The cool-lex order for multiset permutations [Wil09b], discussed in Section 3.4, provides the recursive structure of our left-shift Gray code of fixed-content Łukasiewicz words. This section discusses the terminology used in describing the cool-lex order for multiset permutations.

Tails and Scuts

Given a multiset S of cardinality n, we define the *tail of length* ℓ to be smallest ℓ symbols arranged in a string in non-increasing order. Formally,

$$\text{tail}(\ell) = t_\ell \cdot t_{\ell-1} \cdots t_2 \cdot t_1, \tag{7.2}$$

where $\text{tail}(n) = t_n \cdot t_{n-1} \cdots t_1$ is the unique non-increasing string with content S.

In English, a *scut* is a short tail. We use the term for a tail that is truncated by the addition of a large first symbol. More specifically, a scut of length ℓ and a tail of length ℓ are identical, except for their first symbol, and the first symbol is larger in the scut. Formally, the *scut of length* $\ell + 1$, with respect to S is

$$\text{scut}(s, \ell) = s \cdot \text{tail}(\ell), \tag{7.3}$$

where $s \in S$ is greater than the first symbol $\text{tail}(\ell + 1)$. We refer to a scut of the form $\text{scut}(s, \ell)$ as an *s-scut*.

Recursive Order

Now we define $\text{cool}(S)$ to be an order of $\mathcal{L}(S)$. More broadly, we define $\text{cool}(S)$ on any multiset S with non-negative symbols whose sum is at least as large as its cardinality, and we henceforth refer to these S as *valid*. We define $\text{cool}(S)$ recursively by grouping the strings with the same scut together. Specifically, the scuts are ordered as follows:

- The scuts are first ordered by their first symbol in increasing order. In other words, s-scuts are before $(s + 1)$-scuts.
- For a given first symbol, the scuts are ordered by decreasing length. In other words, longer s-scuts come before shorter s-scuts.
- The string $\text{tail}(n)$ is the only string without a scut, and it is ordered last.

For example, the rightmost column of Figure 7.1 illustrates this order. More specifically, the scuts appear in the following order:

$$100, 10, 1, 2000, 200, 20, 31000, 3000, 300, \tag{7.4}$$

with the single string $\text{tail}(n) = 321000$ appearing last. Note that 2, 30 and 3 are absent from (7.4) because there are no Łukasiewicz words with these suffixes.

In each scut group the strings are ordered recursively. In other words, the common scut is removed from the strings in a particular group, and then they are ordered according to $\text{cool}(S')$, where S' is the valid multiset obtained by removing the symbols of the common scut from S. For example, in Figure 7.1, the strings with scut 1 are ordered according to $\text{cool}(S')$ where $S' = \{3, 2, 1, 0, 0, 0\} - \{1\} = \{3, 2, 0, 0, 0\}$. The base case of the recursion is when $S = \emptyset$.

In the following subsection it will be helpful to know the first string that has an s-scut. By our recursive order, we know that it will have a longest s-scut. Moreover, the exact string can be obtained from the tail by a single shift. To illustrate this, consider the list in Figure 7.1, and let $\alpha = \text{tail}(n) = 321000$.

- The first string with a 1-scut is $\text{left}_\alpha(4, 2) = 302100$.
- The first string with a 2-scut is $\text{left}_\alpha(3, 1) = 132000$.

- The first string with a 3-scut is $\text{left}_\alpha(2,1) = 231000$.

In other words, the first string with a 1-scut is obtained by shifting a 0 into the second position, while the first strings with 2-scuts and 3-scuts are obtained by shifting 1 and 2 into the first position, respectively. This point is stated more generally in the following remark.

Remark 7.1. *Let S be a valid multiset, and $\text{tail}(n) = t_n \cdot t_{n-1} \cdots t_1$ with $t_i > t_{i-1}$. The first string in $\text{cool}(S)$ with a t_i-scut is $\text{left}_{\text{tail}(n)}(n-i+2,1)$ if $t_{i-1} = 0$ or $\text{left}_{\text{tail}(n)}(n-i+2,2)$ if $t_{i-1} > 0$.*

Less technically, Remark 7.1 says that the first string with a t_i-scut is obtained by shifting the next smallest symbol from its position in $\text{tail}(n)$ into the first position, or second position if it is 0.

7.2.2 Equivalence

Now we prove that the successor rule (7.1) correctly provides the next string in $\text{cool}(S)$. This simultaneously proves that (7.1) is a successor rule for a left-shift Gray code of $\mathcal{L}(S)$, and that $\text{cool}(S)$ is a recursive description of the same.

Theorem 7.1. *Let S be a multiset of non-negative values with cardinality n and sum $\Sigma S = n$. Also, let $\alpha \in \mathcal{L}(S)$ be a Lukasiewicz word with content S, and $\beta \in \mathcal{L}(S)$ be the next string in $\text{cool}(S)$ taken circularly (i.e., if α is the last string in $\text{cool}(S)$, then β is the first string in $\text{cool}(S)$). Then $\beta = \text{left}_\alpha(j,i)$. In other words, the successor rule in (7.1) transforms α into β with a left-shift.*

Proof. Let $\alpha = a_1 \cdot a_2 \cdots a_n$ and $\rho = a_1 \cdot a_2 \cdots a_m$ be α's non-increasing prefix.

- If $m = n$, then $\alpha = \text{tail}(n)$ and it is the last string in $\text{cool}(S)$. We also know that $\mathsf{next}(\alpha) = \text{left}(n,2)$ by (7.1a). This gives the first string in $\text{cool}(S)$ with a 1-scut by Remark 7.1, which is the first string in $\text{cool}(S)$ as expected. This is the only case where (7.1a) is used.
- If $m = n-1$, then α's non-increasing prefix extends until its second-last symbol. Furthermore, we know that $a_n = 1$, since this is the only non-zero value that can appear in the rightmost position. We also know that $\mathsf{next}(\alpha) = \text{left}(m+1,1) = \text{left}(n,1)$ by (7.1b). Thus, Remark 7.1 implies that β is the first string with an x-scut, where x is the smallest symbol larger than 1 in S. This is expected since α is the last string in the order with a 1-scut.

The remaining cases are handled cumulatively (i.e., each assumes that the previous do not hold). Note that $\alpha = \rho \cdot a_{m+1} \cdot a_{m+2} \cdots a_n$ is the last string with $\text{scut}(a_{m+1}, \ell) = a_{m+1} \cdot a_{m+2} \cdots a_w$ in a sublist $\text{cool}(S - \{a_{w+1}, a_{w+2}, \ldots, a_n\})$. We also view $\text{left}_\alpha(j,i)$ in two steps: a_j is left-shifted until it joins the non-increasing prefix, then further to index i. This allows us to use Remark 7.1.

- If $a_m < a_{m+2}$, then the scut at this level of recursion, namely $\text{scut}(a_{m+1}, \ell)$, cannot be shortened since $\ell = 0$. So the next scut will be the longest scut with the next largest symbol, which is true by Remark 7.1 and $\mathsf{next}(\alpha) = \text{left}(m+1,1)$ by (7.1b).
- If $a_{m+2} = 0$ and $\Sigma\rho = m$, then the scut cannot be shortened since the sum of the symbols before the shorter scut will be less than their cardinality. Thus, the next scut will be the longest scut with the next largest symbol, which is true by Remark 7.1 and $\mathsf{next}(\alpha) = \text{left}(m+1,1)$ from (7.1b).
- If $a_{m+2} \neq 0$, then the scut at this level of recursion can be shortened to $\text{scut}(a_{m+1}, \ell - 1)$. Given this shorter scut, the order recursively adds new scuts beginning with the first x-scut, where x is the second-smallest remaining symbol. This is true by Remark 7.1 and $\mathsf{next}(\alpha) = \text{left}(m+2,1)$ by (7.1c).
- Otherwise, $a_{m+2} = 0$. This is identical to the previous case, except that $a_{m+2} = 0$. Thus, Remark 7.1 gives $\mathsf{next}(\alpha) = \text{left}(m+2,2)$ by (7.1d)

Therefore, (7.1) gives the next string in the order, which completes the proof. □

Chapter 8

Loopless Łukasiewicz and Motzkin Word Generation

This chapter gives a loopless implementation of the successor rule in 7.1 as well as a simplified implementation of the algorithm for the special case of Motzkin words. Section 8.1 gives a loopless implementation of (7.1) using an integer linked list representation of a Łukasiewicz word. Section 8.2 gives a loopless implemmentation of (7.1) for the special case of Motzkin words. The algorithm for Motzkin words uses an integer array represeentation of Motzkin words and evaluates at most 3 conditionals per generated string.

8.1 Generating Łukasiewicz Words in Linked Lists

Implementing left-shifts in an array-based representation of Łukasiewicz words with unrestricted content would almost certainly require some form of loop to implement left-shifts. In particular, consider the case of $\text{left}_\alpha(1, k)$ where $\alpha = 1, 2, 3, 4, 5, \cdots, k$. This would require shifting each of the first k symbols down one index and therefore require $O(k)$ time. This is not a constant time operation if k is not a constant. In particular, a case like this would necessitate worst-case linear time to perform a single iteration of (7.1). However, using a linked-list of integers to represent Łukasiewicz words, the left-shifts in (7.1) can be implemented looplessly. If pointers to the nodes at positions i and j are present, then $\text{left}(i, j)$ simply removes the node at position i and re-inserts it before the node at position j. Thus, a linked-list representation of Łukasiewicz words may allow for a loopless implementation of the successor rule in 7.1.

8.1.1 Implementation Observations

The algorithm in 8.2 takes advantage of the following observations about equation 7.1:

1. In all cases, $\overleftarrow{\text{luka}}(\alpha)$ shifts a single symbol at most 2 symbols past the first increase in α to either the first or second position in α.

2. Given m, a pointer to α_m and a pre-calculated value of $\sum \rho$, the correct shift to perform can be determined and executed looplessly.

3. Case 7.1a occurs only when α is in descending order and is guaranteed to create an increase at position 2.

4. Shifting a symbol from position $m+2$ preserves the increase at position $m+1$. The increase at position $m+1$ remains the first increase in α unless the shift creates an increase at the front of the string.

5. Shifting a symbol from position $m+1$ creates an increase at position $m+2$ if $\alpha_m < \alpha_{m+2}$. This becomes the new first increase in the string unless the shift creates an increase at the front of the string.

6. In the case where a symbol is shifted from position $m+1$, $\alpha_m >= \alpha_{m+2}$, and the shift does not create an increase at the front of the string, the new first increase is whatever the previous second increase in the string was previously.

   Shifts from position $m+1$ occurr either when

   (a) $\alpha_m < \alpha_{m+2}$: the new first increase is at position $m+2$.

   (b) $m = n-1$: the new first increase is either at the front of the string or does not exist

   (c) $\alpha_m >= \alpha_{m+2}$ and $\alpha_{m+2} = 0$: the new first increase is either at the front of the string or at the previous second increase. Since $\alpha_{m+2} = 0$, if no increase is created at the front of the string, all symbols between α_{m+2} and the new first increase must be zero

7. If shifting creates an increase at the front of the string, the new $\sum \rho$ is equal to α_1

8. If shifting does not create an increase at the front of the string, the new $\sum \rho$ is equal to its prior value plus the value of the symbol that was shifted.

The observation in 6c necessitates keeping track of all increases in α in order to guarantee the ability to determine the new first increase in constant time (i.e., without scanning the string). This is possible in constant time since left-shifting a symbol from position $m, m+1$, or $m+2$ will never affect any increases past index $m+3$. Thus, a stack-like data structure containing pointers to "increase" nodes and the indices at which they occur is maintained throughout the algorithm's execution. This requires order n additional space.

```
typedef struct ll_node {
    int data;
    struct ll_node* prev;
    struct ll_node* next;
} ll_node;

typedef struct inc {
    struct ll_node* node;
    int index;
} inc;
```

(a) struct definitions for linked list Łukasiewicz word representation and the increase stack.

```
void lshift(ll_node* insert, ll_node* shift){
    //remove shift
    ll_node* sprev=shift->prev;
    ll_node* snext=shift->next;
    sprev->next=snext;
    if(snext)
        snext->prev=sprev;

    //insert shift before insert
    ll_node* iprev=insert->prev;
    shift->prev=iprev;
    if(iprev)
        iprev->next=shift;

    shift->next=insert;
    insert->prev=shift;
}
```

(b) Function for left-shifting a linked list node `shift` to before `insert`

Figure 8.1: Functions and structs used for the algorithm in 8.2

Figure 8.2 gives a loopless C implementation of the successor rule in (7.1). The while loop iterates until the Łukasiewicz word contains no increases and therefore is in descending order, which occurs at the end of a non-cyclic ordering. The `inc* incs` is used as a stack which stores pairs of linked list nodes and indices. The `ll_node*` is the linked list node corresponding to the first increase in α.

First, the algorithm pops an increase off of the `incs` stack, storing the increase's index in `m` and the increase's node in `x`. Next, the algorithm determines which node is to be shifted, denoted via `shift_node`. This is accomplished in the main if-else block. The first if statement evaluates the condition in (7.1b) and if it is true sets `shift_node` to `x`,the node at index $m+1$. Otherwise, it sets `shift_node` to `xn`, the node at index $m+2$. In both cases, the algorithm checks the values of the nodes adjacent to the `shift_node` to determine if an increase has been modified or removed by shifting `shift_node`.

```
void luka_ll(ll_node* hd, ll_node* tl, int n, void (*visit)(ll_node* hd, int n)){
    inc* incs = (inc*) calloc(n, sizeof(inc)); //stack of (node, index) pairs
    int nincs=1; //number of increases

    ll_node *shift_node, *insert_node, *x, *xn;
    int prefix_sum,m,insert_index;

    incs[0] = (inc) {.node=tl,.index=n-1}; //cool struct initializer syntax

    while(nincs){
        m = incs[nincs-1].index;
        x=incs[nincs-1].node; //node at index m+1
        xn=x->next; //node at index m+2

        if(m >= n-1 || xn->data > x->prev->data || (prefix_sum == m && xn->data == 0)){
            if(m >= n-1 || xn->data > x->data || xn->data == 0){ //increase removed
                nincs--;
            }else{ //increase kept
                incs[nincs-1].node=x->next;
                incs[nincs-1].index++;
            }
            shift_node=x;
        }
        else{
            shift_node=xn; //shift x+1...
            incs[nincs-1].index++;
            if(xn->next && xn->next->data > xn->data && xn->next->data <= x->data){
                incs[nincs-2] = incs[nincs-1];
                nincs--;
            }
        }

        insert_index=!(shift_node->data); //bang
        if(insert_index){
            insert_node=hd->next;
        }else{
            insert_node=hd;
            hd=shift_node;
        }

        lshift(insert_node,shift_node);
        if(insert_index != m && (shift_node->data < insert_node->data)){
            prefix_sum=hd->data;
            incs[nincs++]= (inc) {.node = insert_node, .index=insert_index+1};
        }else{
            prefix_sum+=shift_node->data;
        }

        visit(hd,n);
    }
}
```

Figure 8.2: Function for looplessly generating all Łukasiewicz words with a fixed set of content.

The next block of code uses the fact that the index to shift to is 1 if a 1 is being shifted and 0 otherwise. The head of the list, `hd`, is updated if a node is shifted to index 0. Finally, the algorithm executes the left-shift with `insert_node` and `shift_node`, checks if an increase was created at the front of the list, and updates the `prefix_sum`. This algorithm is clearly loopless, as `lshift` is a constant time operation and the function `luka_ll` has no inner loops.

8.2 Generating Motzkin Words in Arrays

Since Łukasiewicz words are a generalization of Motzkin words, the same algorithm can be used to generate Motzkin words by restricting the content set S to be strictly zeroes, ones, and twos. However, the additional restrictions on Motzkin words allow for a simpler implementation of the rule. In particular, the content restriction obviates the need for an increase stack. Pseudocode for loopless generation of Motzkin words is given below in Fig. 8.3.

Algorithm 1 Motzkin

function COOLMOTZKIN(s, t)
$n \leftarrow 2 * s + t$
$b \leftarrow 2^1 0^1 2^{s-1} 1^t 0^{s-1}$
$x \leftarrow 3$
$y \leftarrow 2$
$z \leftarrow 2$
`visit`(b)
while $x <= n$ **do**
  $q \leftarrow b_{x-1}$
  $r \leftarrow b_x$
  $b_x \leftarrow b_{x-1}$
  $b_y \leftarrow b_{y-1}$
  $b_z \leftarrow b_{z-1}$
  $b_1 \leftarrow r$
  $x \leftarrow x + 1$
  $y \leftarrow y + 1$
  $z \leftarrow y + 1$
  if $b_x = 0$ **then**
    if $z - 2 > x - y$ **then**
      $b_1 = 2$
      $b_2 = 0$
      $b_x = r$
      $x \leftarrow 3$
      $y \leftarrow 2$
      $z \leftarrow 2$
    else
      $x \leftarrow x + 1$
  else if $q \geq b[x]$ **then**
    $b_x \leftarrow 2$
    $b_{x-1} \leftarrow 1$
    $b_1 \leftarrow 1$
    $z \leftarrow 1$
  `visit`(b)

Figure 8.3: Pseudocode algorithm for loopless enumeration of Motzkin words

Chapter 9

Final Remarks

9.1 Summary

In this thesis, we presented two new Gray codes for generating Catalan objects in cool-lex orders and loopless implementations of each of these Gray codes. Chapter 4 introduced the first pop-push Gray code for generating ordered trees. Chapter 5 provided two loopless implementations of the ordering given in Chapter 4, constituting the first fully published loopless algorithms for generating ordered trees via a pointer-based representation. Chapter 7 gives a shift Gray code for generating Łukasiewicz words with fixed content using a singlt left-shift per generated string. Chapter 8 gives loopless implementation of the ordering in Chapter 7 using a linked list representation of Łukasiewicz words for the general case and a simplified array based implementation for the special case of Motzkin words.

9.2 Open Problems

The results of this thesis have many natural extensions. For example, chapters 4 and 5 examine the cool-lex order for ordered trees. One could imagine generating other Catalan objects in cool-lex order and extending the simultaneous Gray code for Dyck words, binary trees, and ordered trees to more Catalan objects. For example, what the ordering generated by 4 look like if translated to polygon triangulations? What would it look like translated to fixed-length (not fixed content) Łukasiewicz words? We leave these questions as exercises to the reader. Another area of extension for the ordered trees result is the pursuit of a simpler Gray code. The Gray code in chapter 4 uses either one or two pulls per generated tree. Whether a single pull Gray code for ordered trees exists is an open problem.

The loopless algorithm for generating Łukasiewicz words given in section 8.1 could almost certainly be simplified. In particular, an implementation that uses a singly linked list instead of a doubly linked list is likely possible. This would mirror the singly linked list implementation for looplessly generating multiset permutations by Williams [Wil09a], discussed in 3.4.

A further avenue for extension is generalizations of the recursive structure of cool-lex order. As was discussed in section 1.3.1, the binary reflected Gray code has many sublists that are also Gray codes. Beyond this, the idea of reversing sublists of the Gray code has been used in other Gray codes that are not sublists of the binary reflected Gray Code. In particular, Eades and Mckay gave a modification of the binary reflected Gray code which divided strings into 3 recursive cases to generate a homogenous transposition Gray code for (s, t)-combinations. This ordering has a stronger closeness property than generating (s, t) combinations via a sublist of the binary reflected Gray code: successive strings differ by transpositions where bits between the transpositions are all zero. Similarly to the Eades-Mckay modification of the binary reflected Gray code, one could move past sublists of cool-lex order into generalizing the recursive structure of the order. To understand this idea, figure 9.1 shows how co-lexicographic order is transformed into cool-lex order via sublist rotations. More generally, other rotations of cool-lex order could be created to generate new orderings. In other words, one could look for orders that are cooler than cool.

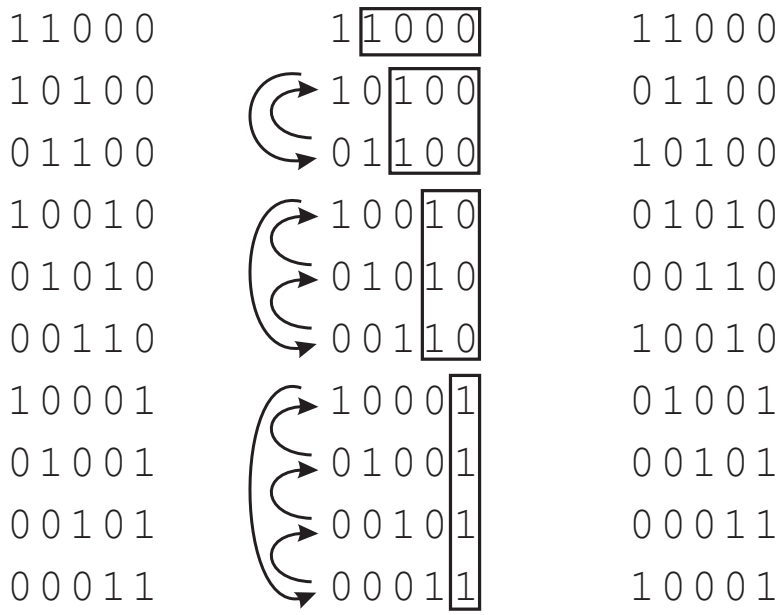


Figure 9.1: Cool-lex order constructed from a rotation of sublists of co-lexicographic order.

Bibliography

[ABP04] Eyal Ackerman, Gill Barequet, and Ron Y Pinter. On the number of rectangular partitions. In *SODA*, pages 736–745, 2004.

[Bob11] Susanne Bobzien. The combinatorics of stoic conjunction. *Oxford Studies in Ancient Philosophy*, 40, 2011.

[Bur20] Madeline Burbage. A hardware engine for generating number-theoretic sequences. 2020.

[CWKB21] James Curran, Aaron Williams, Jerome Kelleher, and Dave Barber. Package 'multicool', Jun 2021.

[DLM+12] Stephane Durocher, Pak Ching Li, Debajyoti Mondal, Frank Ruskey, and Aaron Williams. Cool-lex order and k-ary catalan structures. *Journal of Discrete Algorithms*, 16:287–307, 2012.

[DS77] Robert Donaghey and Louis W Shapiro. Motzkin numbers. *Journal of Combinatorial Theory, Series A*, 23(3):291–301, 1977.

[Er85] MC Er. Enumerating ordered trees lexicographically. *The Computer Journal*, 28(5):538–542, 1985.

[HHMW20] Elizabeth Hartung, Hung P Hoang, Torsten Mütze, and Aaron Williams. Combinatorial generation via permutation languages. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1214–1225. SIAM, 2020.

[KL00] James F Korsh and Paul LaFollette. Multiset permutations and loopless generation of ordered trees with specified degree sequence. *Journal of Algorithms*, 34(2):309–336, 2000.

[Knu15] Donald Ervin Knuth. The art of computer programming: Combinatorial algorithms, vol. 4, 2015.

[KS20] Donald L Kreher and Douglas R Stinson. *Combinatorial algorithms: generation, enumeration, and search*. CRC press, 2020.

[LW22a] Paul Lapey and Aaron Williams. A shift Gray code for Lukasiewicz words. In *International Workshop on Combinatorial Algorithms*, page in press. Springer, 2022.

[LW22b] Paul W Lapey and Aaron Williams. Pop & push: Ordered tree iteration in O(1)-time, 2022. Manuscript submitted for publication.

[MM21] Arturo Merino and Torsten Mütze. Efficient generation of rectangulations via permutation languages. In *37th International Symposium on Computational Geometry (SoCG 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[PM21] Victor Parque and Tomoyuki Miyashita. An efficient scheme for the generation of ordered trees in constant amortized time. In *2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, pages 1–8. IEEE, 2021.

[Rus03] Frank Ruskey. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada*, 11:20, 2003.

[RW05] Frank Ruskey and Aaron Williams. Generating combinations by prefix shifts. In *International Computing and Combinatorics Conference*, pages 570–576. Springer, 2005.

[RW08] Frank Ruskey and Aaron Williams. Generating balanced parentheses and binary trees by prefix shifts. In *Proceedings of the fourteenth symposium on Computing: the Australasian theory-Volume 77*, pages 107–115, 2008.

[Ska88] Wladyslaw Skarbek. Generating ordered trees. *Theoretical Computer Science*, 57(1):153–159, 1988.

[SS00] Louis W Shapiro and Robert A Sulanke. Bijections for the schröder numbers. *Mathematics Magazine*, 73(5):369–376, 2000.

[ST20] Neil J. A. Sloane and The OEIS Foundation Inc. The on-line encyclopedia of integer sequences, 2020.

[Sta97] Richard P Stanley. Hipparchus, plutarch, schröder, and hough. *The American mathematical monthly*, 104(4):344–350, 1997.

[Sta15] Richard P Stanley. *Catalan numbers*. Cambridge University Press, 2015.

[SW09] J Sawada and A Williams. Fixed-density necklaces and lyndon words in cool-lex order. 2009.

[SW12] Brett Stevens and Aaron Williams. The coolest order of binary strings. In *International Conference on Fun with Algorithms*, pages 322–333. Springer, 2012.

[SWW21] Joe Sawada, Aaron Williams, and Dennis Wong. Inside the binary reflected gray code: Flip-swap languages in 2-gray code order. In *International Conference on Combinatorics on Words*, pages 172–184. Springer, 2021.

[VB91] D Roelants Van Baronaigien. A loopless algorithm for generating binary tree sequences. *Information Processing Letters*, 39(4):189–194, 1991.

[Wil09a] Aaron Williams. Loopless generation of multiset permutations by prefix shifts. In *SODA 2009, Symposium on Discrete Algorithms*, 2009.

[Wil09b] Aaron Williams. Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *Proceedings of the twentieth annual ACM-SIAM symposium on discrete algorithms*, pages 987–996. SIAM, 2009.

[Wil09c] Aaron Michael Williams. *Shift gray codes*. PhD thesis, 2009.

[Zak80] Shmuel Zaks. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 10(1):63–82, 1980.