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

Paul Lapey

April 25, 2022

Contents

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].

A quintessential result of the combinatorial generation in practice is Frank Gray's reflected binary code, or Gray code. Gray codes give 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 $\sum_{i=0}^{n} 2^i$, which is 1.9375 for 4 bit values and 1.996 for 8 bit values) bits on average[1]. The binary reflected Gray code, therefore, provides an ordering that requires half as many bit switches on average as the more intuitive lexicographic order. Binary reflected Gray codes are widely used in electromechanical switches to reduce error and prevent spurious output associated with asynchronous bit switches. The technique of reflecting all or certain parts of a string to generate new strings has become one of the most widely used techniques in combinatorial generation. Crucially, Frank Gray's reflected binary code achieved a tangible benefit in error reduction through the use of an alternative method of enumerating binary strings.

[1]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 $\sum_{i=0}^{n} 2^i$, which converges to 2 as n grows large.

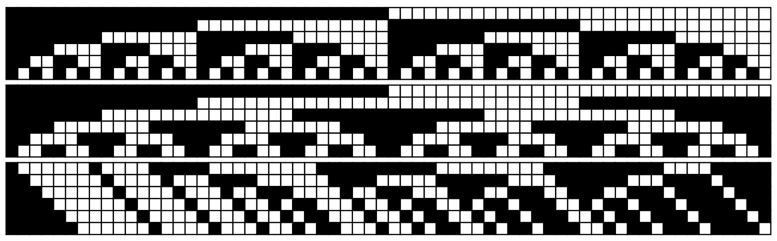


Figure 1.1: Lexicographic (top), binary reflected Gray code (middle), and cool-lex (bottom) enumerations of 6-bit binary strings.
Individual strings are read vertically with the most significant bit at the top; white is 1.

1.2 Gray Codes for Strings, Lattice Paths and Trees

In this thesis, we are not concerned with counting combinatorial objects, but rather with efficiently ordering them. We aim to create a *Gray code* or *minimal change ordering* for these objects. 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. Our Gray codes for strings will use a slightly more restrictive type of shift: a *left-shift*, which moves a single symbol somewhere to the left within a string. More specifically, if $\alpha = a_1a_2 \dots a_n$ is a string and $i < j$, then we let

$$\text{left}_\alpha(j, i) = a_1a_2 \dots a_{i-1}a_ja_ia_{i+1} \dots a_{j-1}a_{j+1}a_{j+2} \dots a_n. \tag{1.1}$$

Our Gray code for ordered trees will 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."

1.3 Cool-Lex Order

Cool-lex order has introduced the idea of rotating sublists to enumerate languages. Different versions of cool-lex order have been shown to enumerate several sets of combinatorial objects, including binary strings, fixed weight binary strings, Dyck words, and multiset permutations [Wil09c]. Cool-lex orders often lead to algorithms that are faster and simpler than standard lexicographic order. 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 versoin 1.1 and as of version 1.12 has been downloaded nearly a million times [CWKB21]. A common thread in the cool-lex algorithms for combinatorial generation is their focus on the *non-increasing prefix* of string, or the longest prefix of a string such that each successive symbol in the prefix is less than or equal to the previous symbol in the string.

1.4 Goals of this Thesis

This thesis will examine the use of cool-lex orders to enumerate other languages. Among these are ordered trees, Lukasiewicz words, and Motzkin words.

This thesis will provide 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.

The second contribution is a shift Gray code for Lukasiewicz words. Chapter 7 will give a shift Gray code for generating Lukasiewicz 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 Lukasiewicz words.

Chapter 2

Catalan Objects

Introduce all these things plus concepts/features that will be useful later. 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 2.1 demonstrates this for the case of $n = 3$ The sequence of Catalan numbers for $n \geq 0$ can be defined mathematically as follows:

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

The Catalan numbers can also be expressed through a summation that hints at the recursive structure of many of the Catalan objects.

$$\mathcal{C}_{n+1} = \sum_{k=0}^{n} \mathcal{C}_k \mathcal{C}_{n-k},\, \mathcal{C}_0 = 1 \tag{2.2}$$

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

$\mathcal{C}_{n+1}$ is the way 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 $C_k * C_{n-k}$ 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 2.2 illustrates this process for the case of $n + 1 = 6$.

The Catalan numbers count a remarkable number of interesting and useful combinatorial objects 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.

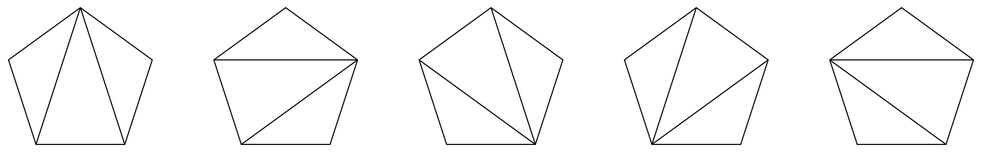
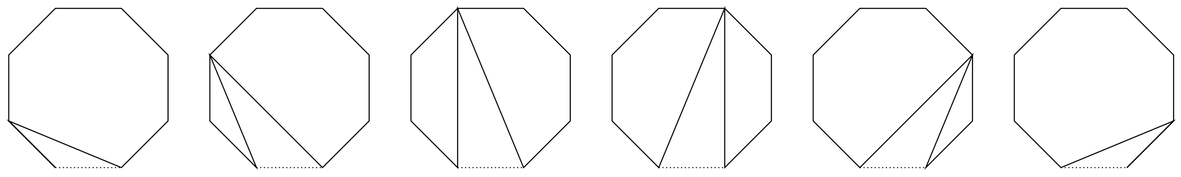


Figure 2.1: The $\mathcal{C}_3 = 5$ triangulations of a polygon with $3 + 2 = 5$ sides.



$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
2-gon; 7-gon	3-gon; 6-gon	4-gon; 5-gon	5-gon; 4-gon	6-gon; 3-gon	7-gon; 2-gon
$\mathcal{C}_0 * \mathcal{C}_5$	$\mathcal{C}_1 * \mathcal{C}_4$	$\mathcal{C}_2 * \mathcal{C}_3$	$\mathcal{C}_3 * \mathcal{C}_2$	$\mathcal{C}_4 * \mathcal{C}_1$	$\mathcal{C}_5 * \mathcal{C}_0$
$1 * 42$	$1 * 14$	$2 * 5$	$5 * 2$	$14 * 1$	$42 * 1$
42	14	10	10	14	42

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

Figure 2.2: Constructing $\mathcal{C}_6$, the number of ways to triangulate an octagon, from triangulations of smaller sub-polygons.

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 $\mathcal{C}_n$. 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 2.3 gives an illustration of each of these interpretations of Dyck words for $n = 4$.

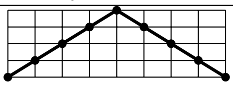
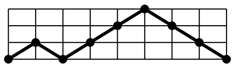
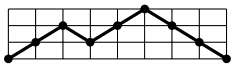
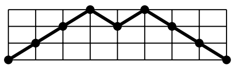
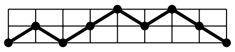
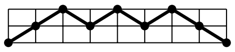
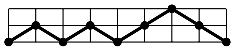
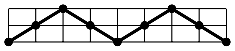
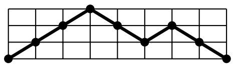
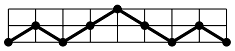
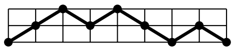
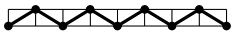
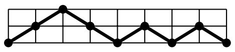
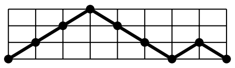
Dyck Path	Dyck Word	Parentheses
	11110000	(((())))
	10111000	()((()))
	11011000	(()(()))
	11101000	((()()))
	10110100	()(()())
	11010100	(()()())
	10101100	()()(())
	11001100	(())(())
	11100100	((())())
	10110010	()(())()
	11010010	(()())()
	10101010	()()()()
	11001010	(())()()
	11100010	((()))()

Figure 2.3: The $\mathcal{C}_4 = 14$ Dyck words of order 4

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: The empty set ϕ is a binary tree. Otherwise a binary tree has a root vertex, a left subtree, and a right subtree, where each subtree is also a binary tree. 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.

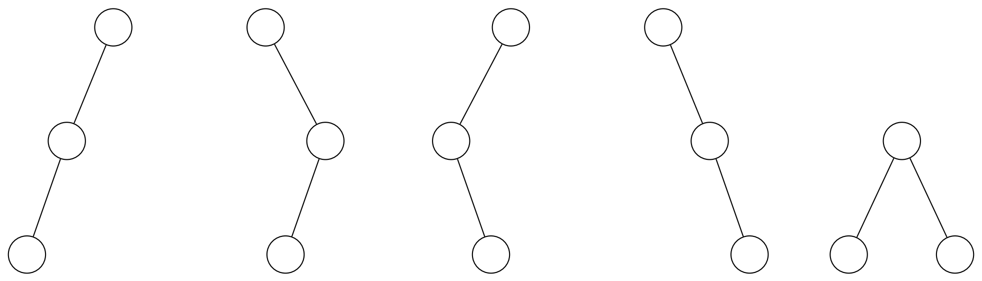
This process can be reversed to construct an extended binary tree with n internal nodes from a binary tree with n nodes. Given a binary tree b with n nodes, add two leaf children to every leaf in b and add one leaf child to every node in b with one child. Following these steps, every node originally in b is now an internal node, and therefore the constructed tree is an extended binary tree with n internal nodes.

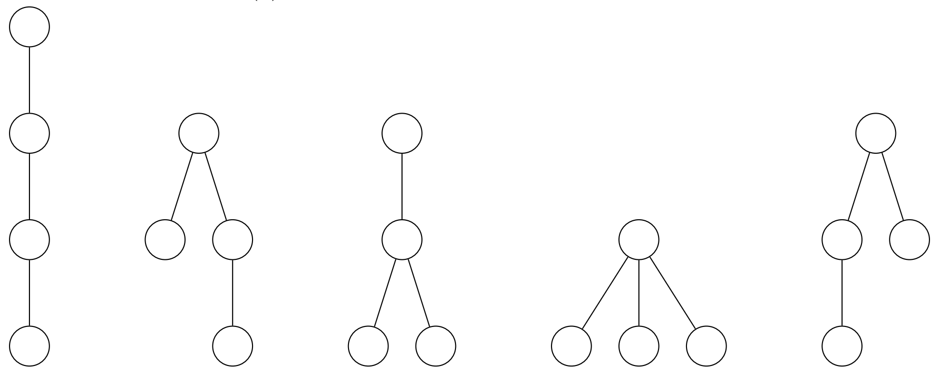
2.3 Ordered Trees

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 $\mathcal{C}_3 = 5$ binary trees with 3 nodes


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

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.

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 $\mathbf{E}_n$.

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 Richard Stanley's *Catalan Objects* [Sta15]. Figure 2.6 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.

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

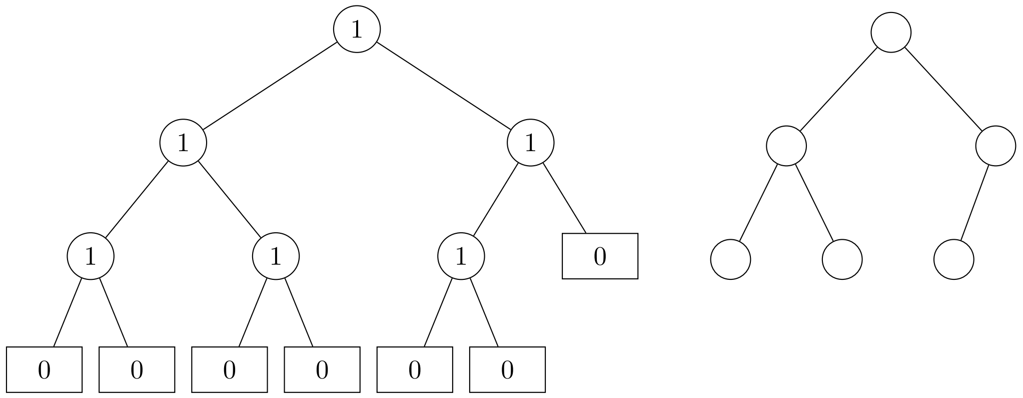


Figure 2.5: The extended binary tree (left) and binary tree (right) corresponding to the Dyck word 111001001100

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

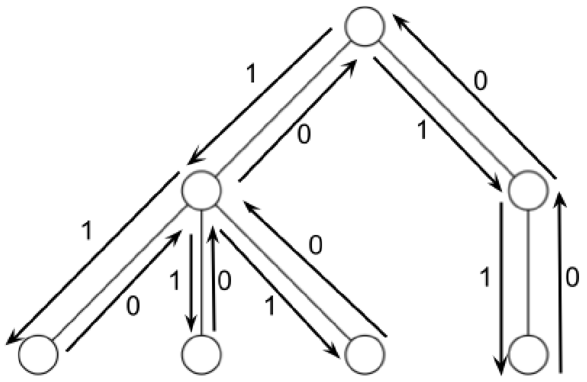


Figure 2.6: An ordered tree with $6 + 1 = 7$ nodes corresponding to the order 6 Dyck word 110101001100.

- For each d_i such that $1 \le i \le 2n$
  - 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.

Chapter 3

Cool-Lex Order

This chapter will give background information on cool-lex order and the sets it has been used to enumerate.

3.1 Combinations: Fixed-Weight Binary Strings

Generating all binary strings with s zeroes and t ones is often referred to as 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 and is as follows [Wil09c]:

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 $x > y$. If there is no such 1, let $x = n$.

Note that $\alpha_1 ... \alpha_{x-1}$ is the non-increasing prefix of α.

Let $\text{left}_\alpha(i)$ be a simplified version of the left shift function defined in equation 1.1: $\text{left}_\alpha(i)$ shifts the $i^{\underline{th}}$ symbol of α into the first position. Thus, $\text{left}_\alpha(i) = \text{left}_\alpha(1, i)$.

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

Note that $\alpha_1 ... \alpha_{x-1}$ must be exactly $1^{y-1}0^{x-y}$, where exponentiation denotes repeated symbols. Because of this, the two left-shift operations can be replaced with can be replaced 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, $\mathsf{transpose}(\alpha, i, j) = \alpha_1, \alpha_2, \ldots, \alpha_{i-1}, \alpha_j \alpha_{i+1} \ldots \alpha_{j-1} \alpha_i \alpha_{j+1} \ldots \alpha_n$ The left-shift rule can be re-stated as follows:

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

3.2 Cool Lex Order on Dyck Paths and Binary Trees

Ruskey and Williams found the following successor rule for enumerating binary Dyck words, dubbed "CoolCat" due to its use of a cool-lex order to generate (cat)alan objects [RW08]: 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 $D \in \mathbf{D}_n$

Let the ith prefix shift of D, denoted by $\mathsf{preshift}_D(i)$, be a function that rotates the second through $i^{\underline{th}}$ symbols of D one to the right circularly. More formally,

$\mathsf{preshift}_D(i) = d_1, d_i, d_2, ..., d_{i-1}, d_{i+1}, d_{i+2}, ..., d_{2n}$

Note that the prefix shift operation is a special case of the left shift operation defined in 1.1: $\mathsf{preshift}_D(i) = \text{left}_D(2, i)$

Let k be the index of the 1 in the leftmost 01 substring in D if it exists and $2n$ if D has no 01 substring. Note that if D has no 01 substring, then $D = 1^n 0^n$. Furthermore, note that this definition of k is equivalent to the definition of x in 3.1 The successor rule for D is as follows:

$$\overleftarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}_D(k+1) & \text{if } \mathsf{preshift}_D(k+1) \in \mathbf{D}_n \quad (3.1a) \\ \mathsf{preshift}_D(k) & \text{otherwise} \quad (3.1b) \end{cases}$$

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 slightly 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.

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

If $D_{k+1} = 1$, then shifting it into the second position is valid. If $D_{k+1} = 0$ and D starts with at least $\frac{k-1}{2}$ 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 3.1 can be simplified to the following:

$$\overleftarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}_D(k+1) & D_{k+1} = 1 \text{ or } D \text{ starts with at least } \lfloor \frac{k-1}{2} \rfloor \text{ ones} \quad (3.2a) \\ \mathsf{preshift}_D(k) & \text{otherwise} \quad (3.2b) \end{cases}$$

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

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.

Due to its simplicity and efficienty, 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].

3.3 Multiset Permutations

Cool-lex order has also been shown to enumerate multiset permutations via prefix shifts. The rule given by Williams is as follows [Wil09a]:

Let α be a multiset of length n.

Let i be the maximum value such that $\alpha_{j-1} \geq s_j$ for all $2 \leq j \leq i$. In other words, i is the length of the non-increasing prefix of α.

Recall the definition of $\text{left}_\alpha(i)$ from section 3.1

Then

$$\text{nextPerm}(\alpha) = \begin{cases} \text{left}_\alpha(i+1) & \text{if } i \leq n-2 \text{ and } \alpha_{i+2} > \alpha_i \\ \text{left}_\alpha(i+2) & \text{if } i \leq n-2 \text{ and } \alpha_{i+2} \leq \alpha_i \\ \text{left}_\alpha(n) & \textit{otherwise} \end{cases}$$

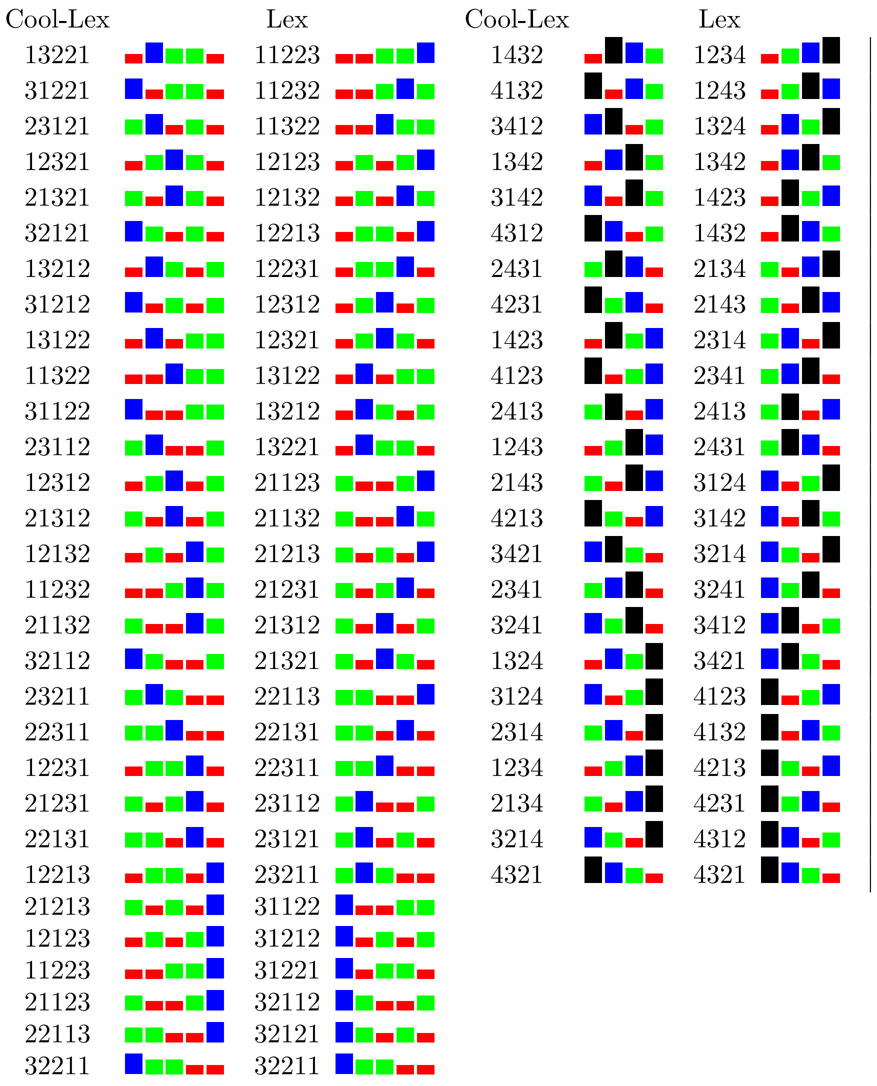
See Fig. 3.1 for an example comparison of cool-lex and lexicographic order for two multisets.

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 allows for a loopless implementation of the successor rule, 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 Lukasiewicz words in chapter 8.



Cool-Lex	Lex	Cool-Lex	Lex
13221	11223	1432	1234
31221	11232	4132	1243
23121	11322	3412	1324
12321	12123	1342	1342
21321	12132	3142	1423
32121	12213	4312	1432
13212	12231	2431	2134
31212	12312	4231	2143
13122	12321	1423	2314
11322	13122	4123	2341
31122	13212	2413	2413
23112	13221	1243	2431
12312	21123	2143	3124
21312	21132	4213	3142
12132	21213	3421	3214
11232	21231	2341	3241
21132	21312	3241	3412
32112	21321	1324	3421
23211	22113	3124	4123
22311	22131	2314	4132
12231	22311	1234	4213
21231	23112	2134	4231
22131	23121	3214	4312
12213	23211	4321	4321
21213	31122		
12123	31212		
11223	31221		
21123	32112		
22113	32121		
32211	32211		

Figure 3.1: Illustration comparing cool-lex and lexicographic order for permutations of the multisets with content {1,1,2,2,3} and {1,2,3,4}

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]. Further information on the package is available here: https://www.rdocumentation.org/packages/multicool/versions/0.1-12

Chapter 4

A Pop-Push Gray Code for Ordered Trees

This chapter presents the first loopless algorithm for generating all ordered trees with n nodes.

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

This thesis provides a new algorithm that generates ordered trees with a fixed number of nodes in a cool-lex order. The algorithm generates a minimal change ordering of ordered trees in the same order as their corresponding Dyck words in Ruskey and Williams's paper. Like the cool-lex algorithms for Dyck words and binary trees, this algorithm can be implemented looplessly: each ordered tree takes worst-case constant time to generate. This is faster than other algorithms for generating ordered trees which take constant amortized time [PM21] [Er85] [Zak80] [Ska88]. Moreover, 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.

Parque and Miyashita present a constant amortized time algorithm for generating ordererd trees, claiming that it operates "with utmost efficiency" [PM21]. Our algorithm operates in worst-case constant time per tree, which is faster. To borrow Parque and Miyashita's terminology, perhaps we should say that our algorithm operates with *utmoster* efficiency

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 a noable 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 to traverse the tree is not necessary.

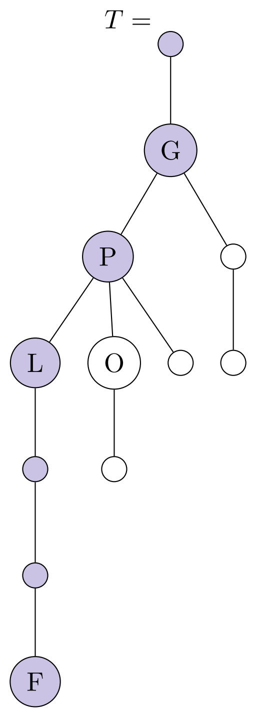
4.1 Successor Rule

Let F be the leftmost leaf of T, or equivalently the leftmost descendant of the root. Consider the unique path between the root of T and F, denoted $\mathsf{path}(\mathsf{T}, \mathsf{root}, \mathsf{F})$. We will refer to this path as the left-down path of T, or $\mathsf{leftpath}(T)$

Given an ordered tree T, let O be the first node in a preorder traversal of T that is not in the $\mathsf{path}(\mathsf{T}, \mathsf{root}, \mathsf{F})$. If $\mathsf{path}(\mathsf{T}, \mathsf{root}, \mathsf{F}) = T$, i.e. the entire tree is a single path, let O be the leaf of the tree. Let P be O's parent. Let G be P's parent, and let L be P's leftmost child (or, equivalently, O's left sibling). The labels P, G, and L are mnemonics for O's (p)arent, (g)randparent, and (l)eft sibling. Fig. 5.2 gives an example illustrating O,P,G,L,F, and the left-down path in an tree.

Given an ordered tree T and an ordered tree node A in 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.



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

Figure 4.1: An ordered tree with 12 nodes corresponding to the Dyck word 1111110000110010011000. The left down path of T is highlighted in purple.

For convenience, we will also define $\mathsf{poppush}(A, B) = \mathsf{pushchild}(B, \mathsf{popchild}(A))$, 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 folows:

$$\mathsf{nextree}(T) = \begin{cases} \mathsf{poppush}(P, G); \mathsf{poppush}(P, root) & P \neq root \text{ and O has no children} \quad (4.1a) \\ \mathsf{poppush}(P, O) & \text{otherwise} \quad (4.1b) \end{cases}$$

Figure 4.2 gives a demonstration of the shifts in cases 4.1a and 4.1b

To make the order cyclic, an additional rule can be added, modifying the successor rule to be:

$$\mathsf{nextree}(T) = \begin{cases} \mathsf{poppush}(P, root) & \mathsf{leftpath}(T) = T \quad (4.2a) \\ \mathsf{poppush}(P, G); \mathsf{poppush}(P, root) & P \neq root \text{ and O has no children} \quad (4.2b) \\ \mathsf{poppush}(P, O) & \text{otherwise} \quad (4.2c) \end{cases}$$

TODO: general illustration

The following remarks can be derived from from the definition of the successor role and the nodes O,G,L, and T.

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 $z = (k - s - 1)$; $d_k = 1$

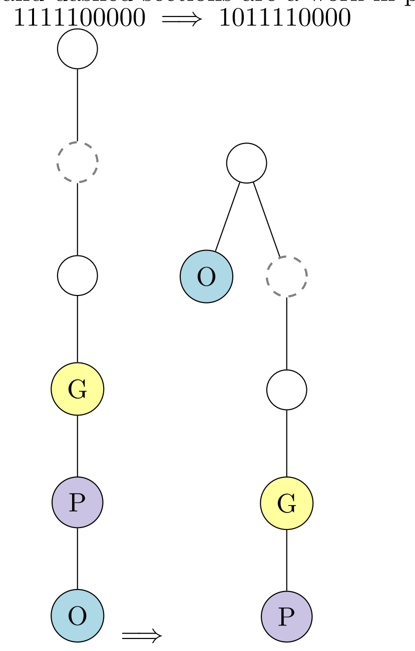
Remark 4.1. $\mathsf{Depth}(O) = s - z + 1$

Proof. t_s is the last node in the $\mathsf{leftpath}(T)$, 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.

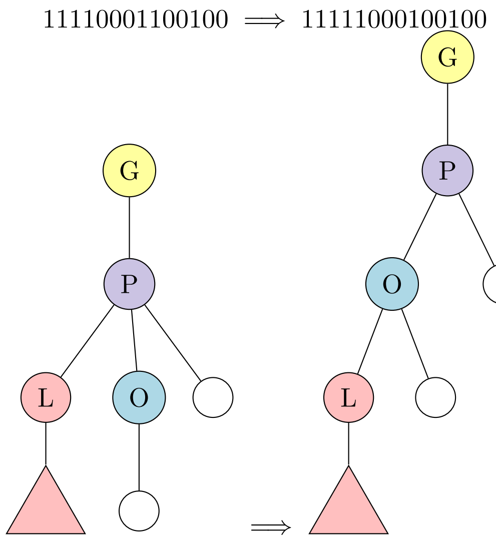
□

Remark 4.2. *O corresponds to* D_k, *i.e.* $\mathsf{oneindex}(D, s + 1) = k$

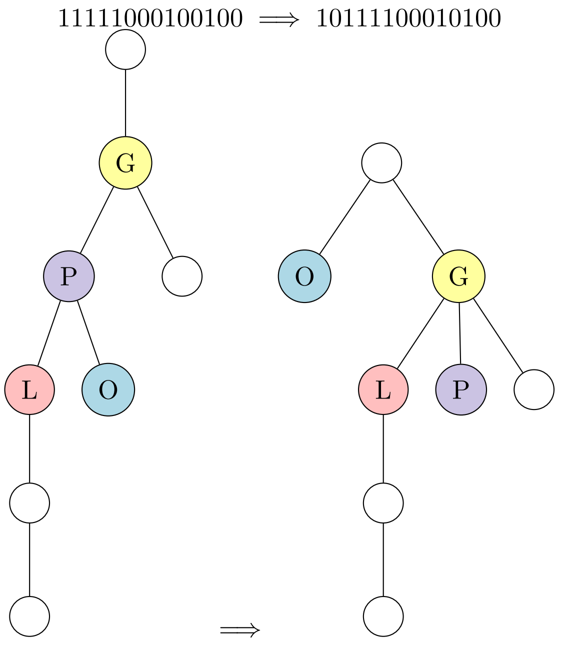
Triangles and dashed sections are a work in progress



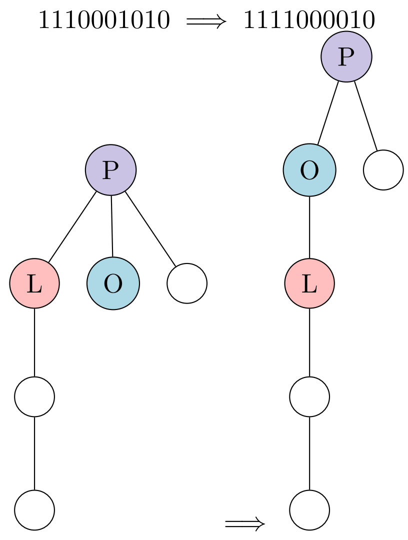
(a) $\mathsf{leftpath}(T) = T$
$\mathsf{poppush}(P, root)$
O becomes the first child of the root



(b) O has at least 1 child
$\mathsf{poppush}(P, O)$
L becomes O's first child



(c) $P \neq root$, O has no children:
$\mathsf{poppush}(P, G)$; $\mathsf{poppush}(P, O)$
L becomes G's first child;
O becomes the first child of $root$



(d) O has no children, $P = root$:
$\mathsf{poppush}(P, O)$
L becomes O's first child

Figure 4.2: Illustrating the successor rule for generating a pop-push Gray code for ordered trees. In these diagrams, O is the first node in a preorder traversal of the tree that is not in the unique path between the root and the tree's leftmost leaf.
P is O's parent, G is P's parent (if it exists), and L is O's left sibling.

Proof. Let $D = \mathsf{Dyck}(T)$ and let k be the index of the 1 in the leftmost 01 substring of D. Let $t_0...t_s = \mathsf{leftpath}(T)$; $O = t_{s+1}$.

Note that each 1 in D corresponds to a step down; each 0 to a step up. Consequently, $\mathsf{leftpath}(T)$ corresponds to the "all-one" prefix of D. In other words, $\mathsf{leftpath}(T) = t_0, t_1, ...t_s$ such that $i = 0$ or $D_i = 1$. Note that t_{s+1} is therefore the first node in a preorder traversal of T such that $D_{\mathsf{oneindex}(D,s+1)} = 1$ and $D_{\mathsf{oneindex}(D,s+1)-1} = 0$. O is therefore also the first node in a preorder traversal of T such that $t_{s+1} \notin \mathsf{leftpath}(T)$. Therfore, $\mathsf{oneindex}(D, s+1) = k$, i.e., $t_{s+1} = O$ corresponds to the 1 in the leftmost 01 substring of D.

□

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

Proof. Suppose by way of contradiction that a node below P in $\mathsf{leftpath}(T)$ had a second child. That child would not be in $\mathsf{leftpath}(T)$ 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 $\mathsf{leftpath}(T)$, which generates a contradiction.

□

Remark 4.4. *L corresponds to* D_{s-z+1} *, i.e.* $\mathsf{oneindex}(D, s - z + 1) = s - z + 1$

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}.

□

4.2 Proof of Correctness

Ruskey and Williams proved that, given a Dyck word of order n, 3.2 iteratively generates all Dyck words of order n. This proof will use the bijection between Dyck words of order n and ordered trees with $n + 1$ nodes to show that that 4.1 generates all ordered trees with a given number of nodes.

4.2.1 Terminology and Remarks

The bijection between ordered trees and Dyck Words given in 2.4.2 has many useful properties for proving the correspondence between 3.2 and 4.1. We will define the following functions to assist in proving this result.

- Let $\mathsf{OTree}(D)$ and $\mathsf{Dyck}(T)$ be functions that convert a Dyck word to an ordered tree and an ordered tree to a Dyck word respectively via the above process.
- Let $\mathsf{Depth}(t_i)$ = length of the path between root and t_i. $\mathsf{Depth}(root) = 0$
- Let $\mathsf{oneindex}(D, i)$ = be the index of the $i^{\underline{th}}$ one in D.

In addition to the above functions, the following remarks can be derived from the bijection between ordered trees and Dyck words. Given an ordered tree T, its preorder listing $t_0, t_1, ...t_n$, and its corresponding order n Dyck word D

Remark 4.5. *The* $i^{\underline{th}}$ *non-root node in a preorder listing of* 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 $T = t_0, t_1, ...t_n$ be a preorder traversal of T. Note that t_0 is the root of 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.

□

4.2.2 Correspondence between coolCat and nextree

Recall that the successor rule $\overleftarrow{\mathsf{coolCat}}(D)$ generates all Dyck words. Therefore, to prove that $\mathsf{nextree}(T)$ 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, $\mathsf{nextree}(T)$ 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,* $\mathsf{nextree}(T) = \mathsf{OTree}(\overleftarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)))$

Proof. $\overleftarrow{\mathsf{coolCat}}(D)$ and $\mathsf{nextree}(T)$ are each broken down into 3 cases in equations 3.2 and 4.1 respectively.

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

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

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

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 $z = (k - s - 1)$; $d_k = 1$

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

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, T is a single path of $n + 1$ nodes, and the left-down path of T is the entire tree. ☐

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

Proof. This follows logically from the bijection between Dyck words and ordered trees. D_k corresponds to O. If $D_{k+1} = 0$, 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. $P = root \iff s = z = \frac{k-1}{2}$.

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 $s + z = k - 1$

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

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

Note that since D has no 01 substring, $D = 1^n 0^n$.

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

$T =$

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

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 $T' = \mathsf{nextree}(T)$,

$T =$

node	t_0	$F = t_s = t_n$	t_1	t_2	$\ldots$	t_{n-1}
depth	0	1	1	2	$\ldots$	$n - 1$
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 T'. Therefore, if D has no 01 substring or $\mathsf{leftpath}(T) = T$,

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

☐

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 $\frac{k-1}{2}$ ones.

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

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

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

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

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

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

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

$T =$

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

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

$T' =$

node	t_0	t_1	$\ldots$	$G = t_{s-z-1}$	$L = t_{s-z+1}$	$\ldots$	$F = t_s$	$P = t_{s-z}$	$O = t_{s+1}$	$\ldots$
depth	0	1	$\ldots$	$(s-z-1)$	$(s-z)$	$\ldots$	$s-1$	$(s-z)$	$(s-z+1)$	$\ldots$
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 T', thereby making P the first node in a preorder traversal of T' that is not in the left-down path of T'. Therefore, $|\mathsf{leftpath}(T')| = s$; $O' = P$.

Recovering a Dyck word from T', we obtain

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

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

$\mathsf{poppush}_{T'}(P, root)$ 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,

$T'' =$

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

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

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

□

Lemma 4.7. *4.4b corresponds to 4.5b*

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

Thus, $\mathsf{nextree}(T)$ 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 $D_{k+1} = 1$,

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

Since $D_{k+1} = 1$, we can rewrite D as. $D = 1^s 0^z 11$

$T =$

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

$\mathsf{poppush}_T(P, O)$ 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, $\mathsf{leftpath}(T)$ will now go through O; every node in $\mathsf{path}(\mathsf{T}, \mathsf{L}, \mathsf{F})$ will have its depth increased by one.

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

$T' =$

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

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 $D = \mathsf{Dyck}(T) = D = 1^s 0^z 11\ldots$ and that $k = s + z + 1$

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

□

Lemma 4.8. *4.4d corresponds to 4.5d*

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

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

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

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

We can thus rewrite $D = \mathsf{Dyck}(T) = 1^s 0^s 101\ldots$

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

Therefore, we can write T as $T =$

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

$\mathsf{poppush}_T(P, O)$ 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, $\mathsf{leftpath}(T)$ will now go through O; every node in $\mathsf{path}(\mathsf{T}, \mathsf{L}, \mathsf{F})$ will have its depth increased by one.

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

$T' =$

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

Since $D = \mathsf{Dyck}(T) = 1^s 0^s 101\ldots$, $\overleftarrow{\mathsf{coolCat}}(D) = 1^{s+1}0^{s+1}1\ldots$ as per case *4.4d*. This is identical to the Dyck word constructed from $T' = \mathsf{nextree}(T)$. 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 $\mathsf{nextree}(T) = \mathsf{OTree}(\overleftarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)))$ in all cases. □

Chapter 5

Loopless Ordered Tree Generation

The algorithm described in this section has two primary implementations: The first implementation uses a linked structure for the nodes, where an ordered tree node stores pointers to its leftmost child and its right sibling. Therefore, to access a node's 3rd child, one would use

```
node->left_child->right_sibling->right_sibling
```

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.

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 an int keeping track of the maximum amount of children it can store (before requiring reallocation)). This implementation stores the list of children "backwards," so to access a node's $k^{\underline{th}}$ child, one would use `node->children[(node->nch)-k]`. Each `node->children[0]` is set to and kept as `NULL` for a null-termination like effect that is useful in the algorithm.

```
typedef struct node {
    int data;
    struct node* parent;
    struct node* left_child;
    struct node* right_sibling;
} node;

node* new_node(int data){
    node* n =
        (node*)malloc(sizeof(node));
    n->data=data;
    n->parent=NULL;
    n->left_child=NULL;
    n->right_sibling=NULL;
    return n;
}
```

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

anode* new_anode(int maxch){
    anode* n =
        (anode*)malloc(sizeof(anode));
    n->maxch=maxch;
    n->parent=NULL;
    n->nch=0;
    n->children =
        (anode**)malloc(sizeof(anode*) * maxch+1);
    n->children[0]=NULL;
    return n;
}
```

Figure 5.1: C code for two different ordered tree structures and functions to initialize a node.

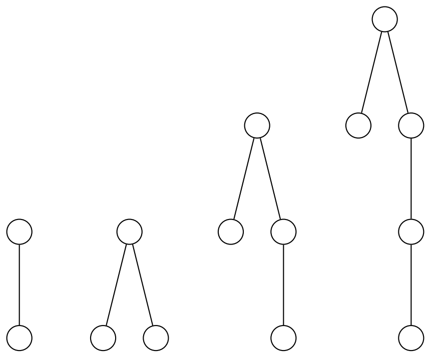


Figure 5.2: The initial trees returned by `get_initial_tree` in `coolOtree` with $t = 1, 2, 3$, and 4
Note the convenient feature that for each tree, O is equal to `root->left_child->right_sibling`

Generate all ordered trees with $t+1$ nodes

```
function COOL-ORDERED-TREES($t$)
    ▷ Generate initial tree
    $O \leftarrow root.lchild$
    visit(root)
    while $O \neq NULL$ do
        $P \leftarrow O.parent$
        if $O.lchild \neq NULL$ then
            pushchild($O$, popchild($P$))
            $O \leftarrow O.lchild.rsibling$
        else
            if $O.parent == root$ then
                pushchild($O$, popchild($P$))
            else
                pushchild($O$, popchild($P$))
                pushchild($root$, popchild($P$))
            $O \leftarrow O.rsibling$
        visit(root)
```

```c
void coolOtree(int t, void (*visit)(node*)){
    node* root = get_initial_tree(t);
    node* o=root->left_child->right_sibling;
    visit(root);
    while(o){
        node* p = o->parent;
        if(o->left_child){
            pushchild(o,popchild(p));
            o=o->left_child->right_sibling;
        }else{
            if(o->parent == root){
                pushchild(o,popchild(p));
            }else{
                pushchild(p->parent,popchild(p));
                pushchild(root,popchild(p));
            }
            o=o->right_sibling;
        }
        visit(root);
    }
}
```

Figure 5.3: Pseudocode and C implementation

Chapter 6

Lattice Paths: Lukasiewicz, Motzkin, Schroder

TODO: this needs to be organized, could use some figures too.

Motzkin, Schröder, and Łukasiewicz paths provide generalizations of Dyck words.

Recall the interpretation of Dyck words as paths in the Cartesian plane from Section 2.1.

Motzkin paths allow for (1,0) horizontal steps in addition to (1,1) and (1,-1) steps. Schröder paths are identical to Motzkin paths except they allow for $(2,0)$ horizontal steps instead of $(1,0)$. Łukasiewicz paths allow (1,-1) steps, (1,0) steps and any (1,k) step where k is a positive integer. 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.

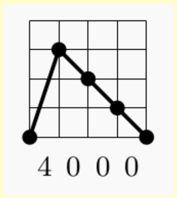
These paths can be encoded in a number of different ways. In a *-1-based encoding*, each $(1,i)$ step is encoded as i, and every prefix must have a nonnegative sum. In a *0-based encoding*, each $(1,i)$ step is encoded as $i+1$, and the sum of every prefix must be as large as its length. We primarily use the 0-based encoding. See Fig. 6.1 for examples of these paths using the 0-based encoding.

We refer to Motzkin, Schröder, and Lukasiewicz paths ending at $(n,0)$ as paths of *order n*. This contrasts slightly with the classification of Dyck words of order n, which terminate at $(2n,0)$

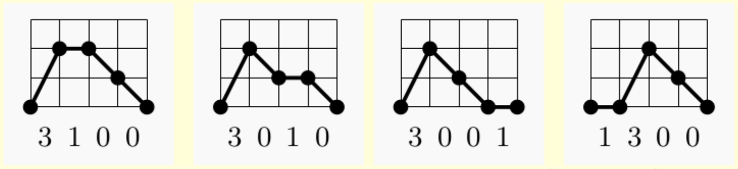
In the context of fixed-content generation, Motzkin and Schröder paths are identical: Both will have northeast steps encoded as twos, horizontal steps encoded as ones, and southeast steps encoded as zeroes. However, their Cartesian plane representations will differ in the length of horizontal steps. Notably, Łukasiewicz are a generalization of Motzkin paths, as any Motzkin path is also a Lukasiewicz path.

The number of Dyck words with n zeroes and n ones are counted by the $n^{\underline{th}}$ Catalan number. Similarly, the number of Motzkin and Schröder paths of order n are counted by the $n^{\underline{th}}$ Motzkin and big Schröder number respectively. The number of Lukasiewicz paths of order n are counted by the $n-1^{\underline{th}}$ Catalan number. Motzkin, Schröder, and Lukasiewicz paths bear a number of interesting bijective correspondences with other combinatorial objects. Richard Stanely's *Catalan Objects* outlines hundreds of interesting examples.

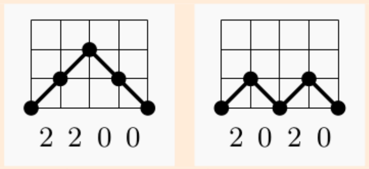
Lukasiewicz paths of order n bear a particularly nice correspondence to rooted ordered trees with n nodes. See Fig. 6.2 for an illustration of this.



The Łukasiewicz paths with content $\{0, 0, 0, 4\}$ using 0-based strings.
Note: The content is {-1, -1, -1, 3} when using (-1)-based meander strings,
or $\{0, 0, 1\}$ for 3-ary Dyck words.



The Łukasiewicz paths with content $\{0, 0, 1, 3\}$.



The Łukasiewicz / Schröder / Motzkin / Dyck paths with content $\{0, 0, 2, 2\}$.
Note: The content is {[, [,],]} for Dyck words or $\{0, 0, 1, 1\}$ for 2-ary Dyck words.

Figure 6.1

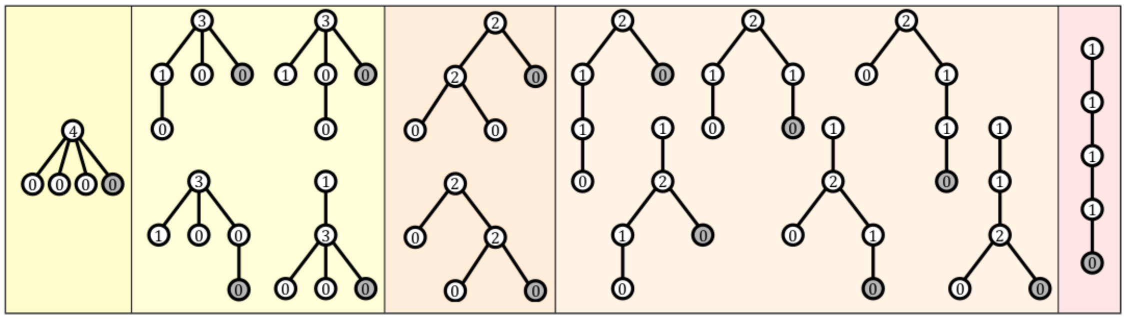


Figure 6.2: The $\mathcal{C}_4$=14 Lukasiewicz paths of order $n = 4$ are in bijective correspondence with the 14 rooted ordered trees with $n + 1 = 5$ nodes. Given a tree, the corresponding word is obtained by recording the number of children of each node in preorder traversal; the zero from the rightmost 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 Lukasiewicz Words

In this chapter, we give a shift gray code for generating Lukasiewicz words with fixed content.

7.1 Successor Rule

In this section, we provide a *successor rule* that applies a left-shift to a Lukasiewicz 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 Lukasiewicz words with content equal to S. Let $\alpha \in \mathcal{L}(S)$.

Recall the definition of a left shift from equation 1.1: $\text{left}_\alpha(i, j)$ shifts the $j^{\underline{th}}$ symbol in α into the $i^{\underline{th}}$ position. In addition to the left shift function, we define the length of the *non-increasing prefix* of a string α to be the maximum value m such that $\alpha_{i-1} \geq \alpha_i$ for all $2 \leq i \leq m$. We will let ρ $\rho = a_1 \cdot a_2 \cdots a_m$. The sum of the symbols in ρ is $\sum \rho = a_1 + a_2 + \cdots + a_m$

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

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 $m = 2$, and the length of the string is $n = 6$. Thus, $m \neq n$, 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

Now we prove that the successor rule is correct. Our strategy is to define a recursive order of $\mathcal{L}(S)$, and show that (7.1) creates the next string in this order.

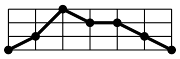
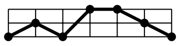
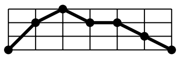
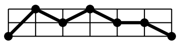
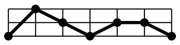
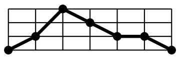
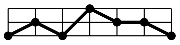
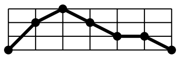
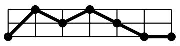
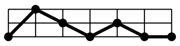
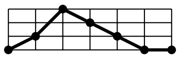
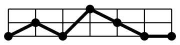
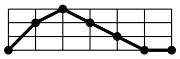
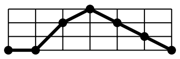
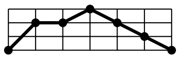
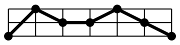
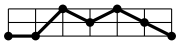
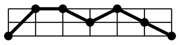
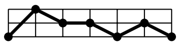
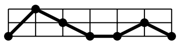
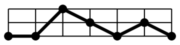
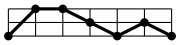
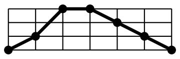
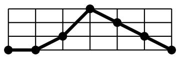
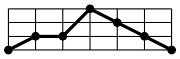
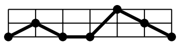
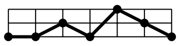
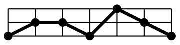
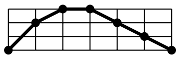
Łukasiewicz path	**Łukasiewicz word**	m	(7.1)	**shift**	**scut**
	302100	2	(7.1b)	left$(3,1)$	100
	230100	1	(7.1d)	left$(3,2)$	100
	203100	2	(7.1b)	left$(3,1)$	100
	320100	3	(7.1d)	left$(5,2)$	100
	302010	2	(7.1d)	left$(4,2)$	10
	300210	3	(7.1b)	left$(4,1)$	10
	230010	1	(7.1d)	left$(3,2)$	10
	203010	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.

7.2.1 Cool-lex Order

Cool-lex order is a variation of co-lexicographic order. The order was first given for (s,t)-*combinations*, which are binary strings with s copies of 0 and t copies of 1, by Ruskey and Williams [RW05, RW09]. In this context, the order gives a *prefix-shift Gray code*, meaning that a single symbol is left-shifted into the first position. The prefix-shift Gray code was then generalized to Dyck words [RW08] and multiset permutations [Wil09b]. The latter result provides the recursive structure of our left-shift Gray code of fixed-content Łukasiewicz words.

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, with 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 Łukasiewicz 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 Lukasiewicz 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 which evaluates at most 3 conditionals per generated string.

8.1 Generating Lukasiewicz Words in Linked Lists

Implementing left-shifts in an array-based representation of Lukasiewicz words would almost certainly require some form of loop left-shifts. In particular, left-shifting a symbol from an position i in the array to front of the array would require shifting each of the first i symbols down. This would necessitate worst-case linear time to perform a single iteration. However, using a linked-list of integers to represent Lukasiewicz words, left-shifts can be implemented looplessly as performing a left-shift requires only removing the node to be shifted from the list it and re-inserting it at position 1 or 2. Thus, a linked-list representation of Lukasiewicz words allows for a loopless implementation of the successor rule in 7.1.

8.1.1 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.

This final observation 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.

——————————work in progress———————————

```
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 Lukasiewicz word representation and the increase stack.

```
void lshift_ll(ll_node* insert_node, ll_node* shift_node){
    //remove shift_node
    ll_node* sprev=shift_node->prev;
    ll_node* snext=shift_node->next;
    if(sprev){
        sprev->next=snext;
    }
    if(snext){
        snext->prev=sprev;
    }

    //insert shift_node before insert_node
    ll_node* iprev=insert_node->prev;
    shift_node->prev=iprev;
    if(iprev){
        iprev->next=shift_node;
    }

    shift_node->next=insert_node;
    insert_node->prev=shift_node;
}
```

(b) helper function for left-shifting a linked list node

```
void luka_ll(ll_node* hd, ll_node* tl, int n, void (*visit)(ll_node* hd)){
    inc* incs = (inc*) calloc(n/2, sizeof(inc)); //stack of (node, index) pairs
    int nincs=1; //number of increases
    incs[0] = (inc) {.node=tl,.index=n-1}; //cool struct initializer syntax

    ll_node *shift_node, *insert_node;
    int prefix_sum,prefix_len,insert_index;

    while(nincs){
        ll_node* m=incs[nincs-1].node;
        prefix_len = incs[nincs-1].index;
        if(prefix_len >= n-1){ // increase removed
            nincs--;
            shift_node=m;
        }
        else if(m->next->data > m->prev->data){
            if(m->next->data > m->data){ //increase removed
                nincs--;
            }else{ //increase kept
                incs[nincs-1].node=m->next;
                incs[nincs-1].index++;
            }
            shift_node=m;
        }
        else{
            if(prefix_sum > prefix_len || m->next->data > 0){ //not tight
                shift_node=m->next; //shift m+2...
                incs[nincs-1].index++; //same increase, different location (shifted down by one)
                if(prefix_len < n-2 && m->next->next->data > m->next->data &&
                 m->next->next->data <= m->data){ //hideous line of code
                    incs[nincs-2] = incs[nincs-1];
                    nincs--;
                }
            }else{ //tight; increase removed
                nincs--;
                shift_node=m;
            }
        }

        insert_index=!(shift_node->data); //bang
        if(insert_index){
            insert_node=hd->next;
        }else{
            insert_node=hd;
            hd=shift_node;
        }
        lshift_ll(insert_node,shift_node);
        if(insert_index != prefix_len && (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);
    }
}
```

Figure 8.2: work in progress. At least it already fits on one page.

8.2 Generating Motzkin Words in Arrays

Since Lukasiewicz 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. 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

9.2 Open Problems

Bibliography

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

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

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

[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.

[RW09] Frank Ruskey and Aaron Williams. The coolest way to generate combinations. *Discrete Mathematics*, 309(17):5305–5320, 2009.

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

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

[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.