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

Paul Lapey

April 4, 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 as many bit switches 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. 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. 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.

---

[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.
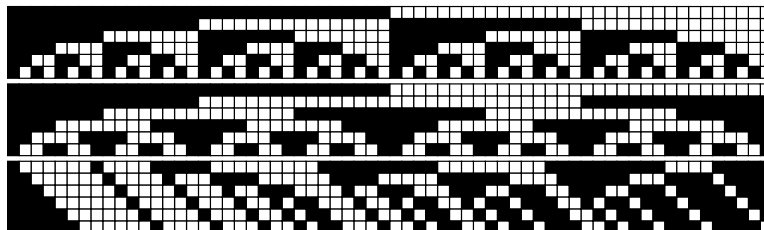

Figure 1.1: Lexicographic (top), binary reflected Gray code (middle), and cool-lex (bottom) enumerations of 6-bit binary strings.
Individual strings are read vertically with the most significant bit at the top; white is 1.

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 $\alpha = a_1 a_2 \ldots a_n$ is a string and $i < j$, then we let

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

Our gray code for ordered trees will use *firstborn-shifts*, which shift the first child of one node to be the first child of another node.

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. 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].

1.4   Goals of this Thesis

Cool-lex has been shown to provide a minimal-change cyclic ordering for the sets of fixed-weight binary strings, multiset permutations, binary and k-ary Dyck words, and other languages [Wil09]. A common thread in the cool-lex algorithms for combinatorial generation is their focus on the *first increase* 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.

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

Dyck, Motzkin, and Lukasiewicz paths all share bijections with various combinatorial objects. For example, Dyck paths of length $2n$ share a bijection with binary trees with n nodes. Ruskey and Williams found that the cool-lex successor rule for enumerating Dyck words corresponded directly to a loopless successor rule for enumerating binary trees with a constant number of pointer changes [RW08].

Chapter 2

Catalan Objects

The Catalan numbers are possibly the most ubiquitous sequence 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. The Catalan numbers can also be defined mathematically as follows:

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

The Catalan numbers count a remarkable number of interesting and useful combinatorial objects in bijective correspondence with triangulation 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 including a thorough history on the numbers and their study [Sta15]. This thesis will focus on three Catalan objects: Dyck words, binary trees, and ordered trees.

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 $\mathcal{C}_n$.

Two common interpretations of Dyck words are in terms of balanced parentheses and in terms of paths in the Cartesian plane. If each one in a Dyck word is taken to represent an open parenthesis and each 0 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 $(0, 0)$ and never go below the x axis. In this case, each 1 in a Dyck word represents a $(1, 1)$ step and each 0 represents a $(1, -1)$ step.

Figure 2.2 gives an illustration of each of these interpretations of Dyck words for $n = 4$.

2.2 Binary Trees

The language

This will be a figure of polygons, someday.

Figure 2.1: Triangulated Polygons!
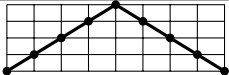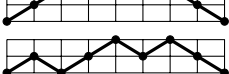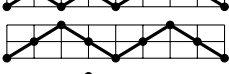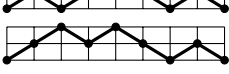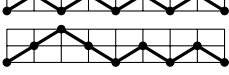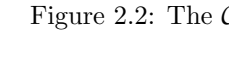
Dyck Path	Dyck Word	Parentheses
	11110000	$((((()))))$
	10111000	$()((()))$
	11011000	$(()(()))$
	11101000	$((()()))$
	10110100	$()(()())$
	11010100	$(()()())$
	10101100	$()()(())$
	11001100	$(())(())$
	11100100	$((())())$
	10110010	$()(())()$
	11010010	$(()())()$
	10101010	$()()()()$
	11001010	$(())()()$
	11100010	$((()))()$

Figure 2.2: The $\mathcal{C}_4 = 14$ Dyck words of order 4
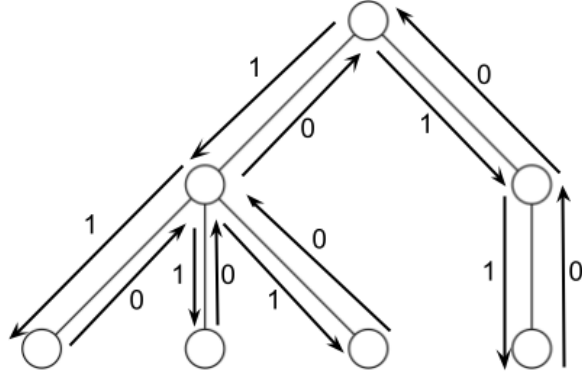
Figure 2.3: An ordered tree with $6 + 1 = 7$ nodes corresponding to the order 6 Dyck word 110101001100.

2.3   Ordered Trees

2.4   Bijections

This algorithm will use the bijection between ordered trees and Dyck words specified in [Sta15]. The bijection described by Stanely is 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:

As before, 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.

Create a root node of T. Keep track of a current node $curr$; set $curr = root$.

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

    - if $d_i = 1$: append a rightmost child ch to $curr$'s children; set $curr = ch$
    - if $d_i = 0$, set $curr$ equal to $curr$'s parent.

Figure 2.3 demonstrates both directions of this process. Note that each t_i with $1 \leq i \leq n$ in a preorder traversal of T corresponds to the ith 1 in D.

In addition to the above bijection, we define the following functions relating to ordered trees, Dyck words, and the correspondence between them.

- 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 path between root and t_i. $\mathsf{Depth}(root) = 0$

- Let $\mathsf{oneindex}(D, i) =$ be the index of the ith one in D.

The following remarks can be derived from the bijection between ordered trees and Dyck words.

Remark 1. *t_i corresponds to the ith one in D for $1 \leq i \leq n$*

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 $1 \leq i \leq n$. $\square$

Remark 2. *The difference in depths between nodes t_i and t_{i-1} is equal to one minus the number of zeroes between the $(i-1)^{st}$ and i^{th} and ones in D*

---

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

$$D = [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0]$$
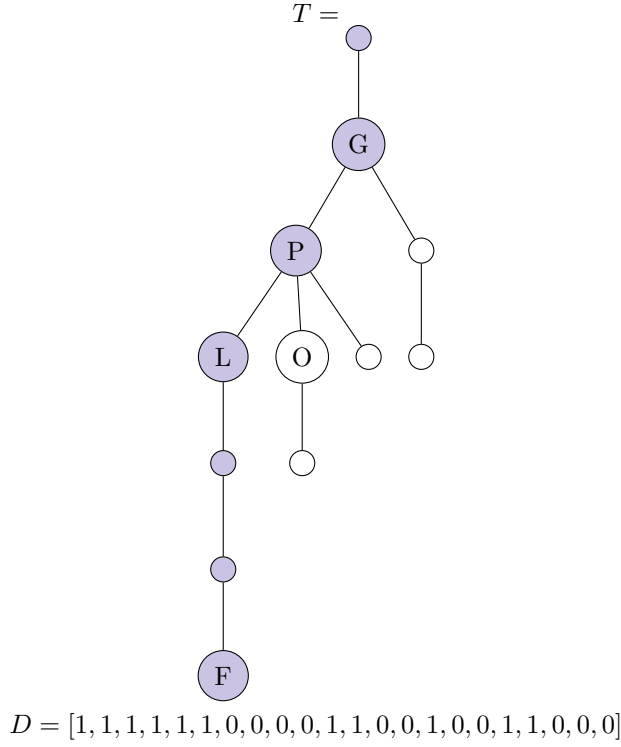
Figure 2.4: An ordered tree with 12 nodes corresponding to the Dyck word 111111000011001001 1000. The left down path of T is highlighted in purple.

Proof. This remark can be stated formally as

$$\mathsf{Depth}(t_i) - \mathsf{Depth}(t_{i-1}) = 1 - (\mathsf{oneindex}(D, i) - \mathsf{oneindex}(D, i - 1) - 1) \tag{2.2}$$

Note that $(\mathsf{oneindex}(D, i) - \mathsf{oneindex}(D, i-1) - 1)$ is equal to the number of zeroes between the ith and $(i-1)^{st}$ 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 ith and $(i-1)^{st}$ 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 ith and $(i-1)^{st}$ 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 3. *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.

□

2.5 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 $\mathbf{B}_n$ to denote binary Dyck words with n ones and n zeroes. Note that the length of any string in $\mathbf{B}_n$ is thus $2n$.

Let $D \in \mathbf{B}_n$

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

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

Let k be the index of the 1 in the leftmost 01 substring in D if it exists. Note that if D has no 01 substring, then $D = 1^n 0^n$. The successor rule for D is as follows:

$$\overrightarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}(D, 2n) & \text{if } D \text{ has no 01 substring} & (2.3a) \\ \mathsf{preshift}(D, k+1) & \text{if } \mathsf{preshift}(D, k+1) \in \mathbf{B}_n & (2.3b) \\ \mathsf{preshift}(D, k) & \text{otherwise} & (2.3c) \end{cases}$$

Ruskey and Williams's algorithm can also enumerate a broader set of strings: The algorithm enumerates any set $\mathbf{B}_{s,t}$ where any $D \in \mathbf{B}_{s,t}$ 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 $\mathbf{B}_n$ languages due to their correspondce with Dyck words and therefore other catalan objects.

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

Let $D' = \mathsf{preshift}(D, k+1)$

Note that we know $D \in \mathbf{B}_n$.

Since preshift only rotates symbols, D' will automaticallly satisfy the requirement that strings in $\mathbf{B}_n$ must have an equal number of zeroes and ones since D satisfied that requirement. Thus, $D' \in \mathbf{B}_n$ will be determined by whether or not all prefixes of D' have at least as many ones as zeroes.

If D_{k+1} is a 1, then for all i, the $i^{\underline{th}}$ prefix of D' will have at least as many ones as the $i^{\underline{th}}$ prefix of D. Thus, D' must be $\in \mathbf{B}_n$, as rotating a 1 to earlier in the string will never invalidate the requirement that every prefix of the string has at least as many ones as zeroes.

Note that the $k^{\underline{th}}$ prefix of D must be of the form $1^a 0^b 1$, as otherwise there would be an earlier 01 prefix. Fruthermore, $a \geq b$ as otherwise the bth prefix of D would have more zeroes than ones and D would not be a valid Dyck word.

If D_{k+1} is a 0, then $D' \notin \mathbf{B}_n$ if and only if rotating a 0 to index 2 creates a prefix of D with more zeroes than ones. This will only happen if the $(k-1)^{\underline{th}}$ prefix of D is exactly $1^{\frac{k-1}{2}} 0^{\frac{k-1}{2}}$.

Therefore, $\mathsf{preshift}(D, k+1) \in \mathbf{B}_n \iff D_{k+1} = 1$ or D starts with more than $\lfloor \frac{k-1}{2} \rfloor$ ones

$$\overrightarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}(D, 2n) & \text{if } D \text{ has no 01 substring} & (2.4a) \\ \mathsf{preshift}(D, k+1) & D_{k+1} = 1 \text{ or } D \text{ starts with more than } \lfloor \frac{k-1}{2} \rfloor \text{ ones} & (2.4b) \\ \mathsf{preshift}(D, k) & \text{otherwise} & (2.4c) \end{cases}$$

Since k is the index of the first 01 substring in D, $\sum_{i=1}^{k} S_i$ is actually just the number of consecutive ones to start D, which simplifies the evaluation of this conditional even further.

Ruskey and Williams provided a loopless pseudocode implementation of CoolCat that utilized this fact to enumerate any $\mathbf{B}_{s,t}$ using at most 2 conditionals per successor [RW08].

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].

Chapter 3

Loopless Ordered Tree Generation

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.

3.1 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. 3.2 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.

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.

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

(b) O has at least 1 child
$\mathsf{poppush}(P,root)$
L becomes O's first child
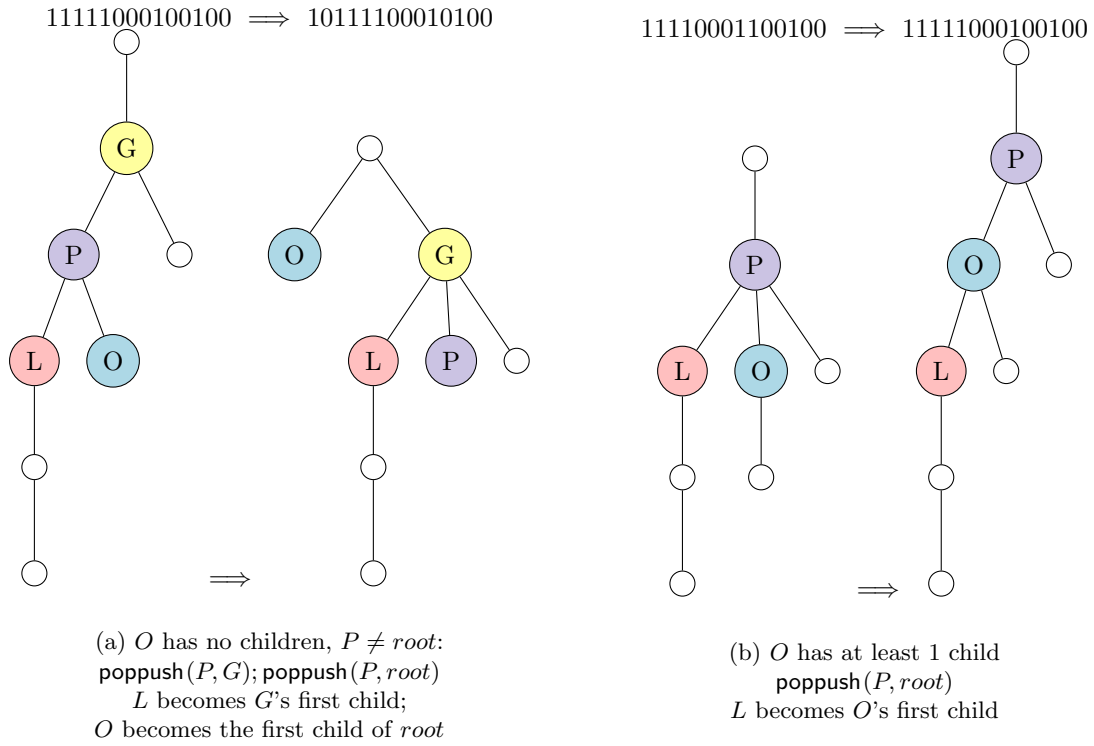
Figure 3.1: Illustrations of cases 3.1a and 3.1b

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

$$\mathsf{nextree}(T) = \begin{cases} \mathsf{poppush}(P,G); \mathsf{poppush}(P,root) & P \neq root \text{ and O has no children} \quad (3.1a) \\ \mathsf{poppush}(P,O) & \text{otherwise} \quad (3.1b) \end{cases}$$

Figure 3.1 gives a demonstration of the shifts in cases 3.1a and 3.1b
To make the order cyclic, an additional rule can be added, modifying the successor rule to be:

$$\mathsf{nextree}(T) = \begin{cases} \mathsf{poppush}(P,root) & \mathsf{leftpath}(T) = T \quad (3.2a) \\ \mathsf{poppush}(P,G); \mathsf{poppush}(P,root) & P \neq root \text{ and O has no children} \quad (3.2b) \\ \mathsf{poppush}(P,O) & \text{otherwise} \quad (3.2c) \end{cases}$$

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

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. $\qquad \square$

Remark 5. *O corresponds to D_k, i.e. $\mathsf{oneindex}(D, s+1) = k$*

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.

$\square$

Remark 6. *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.

$\square$

Remark 7. *L corresponds to* D_{s-z+1}, *i.e.* $\mathsf{oneindex}(D, s+1) = k$

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}.

$\square$

3.2 Proof of Correctness

Ruskey and Williams proved that, given a Dyck word of order n, 2.4 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 3.1 generates all ordered trees with a given number of nodes.

Recall that the successor rule $\overrightarrow{\mathsf{coolCat}}(D)$ 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,

Theorem 3.2.1. *Given an ordered tree T,* $\mathsf{nextree}(T) = \mathsf{OTree}(\overrightarrow{\mathsf{coolCat}}(\mathsf{Dyck}(T)))$

Proof. $\overrightarrow{\mathsf{coolCat}}(D)$ and $\mathsf{nextree}(T)$ are each broken down into 3 cases in equations 2.4 and 3.1 respectively.

For convenience, equations 3.3 and 3.4 give the expanded restatemtents of the successor rules for $\mathsf{nextree}(T)$ and $\overrightarrow{\mathsf{coolCat}}(D)$ to facilitate comparisons between the two.

$$\mathsf{nextree}(T) = \begin{cases} \mathsf{poppush}(P, root) & \mathsf{leftpath}(T) = T & (3.3a) \\ \mathsf{poppush}(P, O) & \text{if O has at least 1 child} & (3.3b) \\ \mathsf{poppush}(P, G); \mathsf{poppush}(P, root) & \text{if } P \neq root \text{ and O has no children} & (3.3c) \\ \mathsf{poppush}(P, O) & \text{if O has no children and } P = root & (3.3d) \end{cases}$$

$$\overrightarrow{\mathsf{coolCat}}(D) = \begin{cases} \mathsf{preshift}(D, 2n) & \text{if } D \text{ has no 01 substring} & (3.4a) \\ \mathsf{preshift}(D, k+1) & D_{k+1} = 1 & (3.4b) \\ \mathsf{preshift}(D, k+1) & D_{k+1} = 0 \text{ and } s > \frac{k-1}{2} & (3.4c) \\ \mathsf{preshift}(D, k) & D_{k+1} = 0 \text{ and } s = \frac{k-1}{2} & (3.4d) \end{cases}$$

We will show the following equivalences:

- 3.3a corresponds to 3.4a

- 3.3b corresponds to 3.4b

- 3.3c corresponds to 3.4c

- 3.3d corresponds to 3.4d

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

Proof. If D has no 01 substring, $D = 1^n0^n$, and T is $n + 1$ nodes where t_0 is the root and each t_i for $1 \le i \le n$ 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. $\square$

Lemma 3.2.3. $D_{k+1} = 0 \iff O$ *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. $\square$

Lemma 3.2.4. $P = root \iff s = z = \frac{k-1}{2}$.

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

Lemma 3.2.5. *3.3a corresponds to 3.4a*

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

Per lemma 3.2.2 D has no 01 substring $\iff$ $\mathsf{leftpath}(T) = T$.

Thus, $\mathsf{nextree}(T)$ executes case 3.3a if and only if $\overrightarrow{\mathsf{coolCat}}(D)$ executes case 3.4a

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

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

$T =$

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 3.

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	...	t_{n-1}
depth	0	1	1	2	...	$n-1$
Dyck		1	$01^{n-1}0^{n-1}$			

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

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

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

$\square$

Lemma 3.2.6. *3.3c corresponds to 3.4c*

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

Per lemma 3.2.4 $P = root \iff D$ 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 3.1a if and only if $\overrightarrow{\mathsf{coolCat}}(D)$ executes case 3.4c

We now show that the execution of 3.1a is equivalent to the execution of 3.4c given case a. Given $\mathsf{Dyck}(T) = D = 1^s 0^z 10 d_{k+2} d_{k+3} ... d_{2n}$, we aim to show that

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

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'=$1^{s-1} 0^z 110 d_{k+2} d_{k+3}, \ldots, d_{2n}$

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 $\mathsf{Dyck}(T) = D = 1^s 0^z 10 \ldots$ 3.4*b* gives that

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

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

$\square$

Lemma 3.2.7. *3.3b corresponds to 3.4b*

14

Proof. Per 3.2.3, as O has at least 1 child $\iff D_{k+1} = 1$.

Thus, $\mathsf{nextree}(T)$ will execute case 3.3b if and only if $\overrightarrow{\mathsf{coolCat}}(D)$ executes case 3.4b

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

$\mathsf{poppush}_T(P, O)$ 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	...	$G = t_{s-z-1}$	$P = t_{s-z}$	$O = t_{s+1}$	$L = t_{s-z+1}$	...	$F = t_s$	$t_{s+2}\ldots$
depth	0	1	...	$(s-z-1)$	$(s-z)$	$(s-z+1)$	$(s-z+2)$	...	$s+1$	$s-z+2\ldots$
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, $\overrightarrow{\mathsf{coolCat}}(D) = \mathsf{preshift}(D, k+1) = 1^{s+1} 0^z 1 \ldots$, which is the same as the Dyck word resulting from translating $T' = \mathsf{nextree}(T)$ to the Dyck word $1^{s+1} 0^z 1 \ldots$

$\square$

Lemma 3.2.8. *3.3d corresponds to 3.4d*

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

Since $D = \mathsf{Dyck}(T) = 1^s 0^s 101 \ldots$, $\overrightarrow{\mathsf{coolCat}}(D) = 1^{s+1} 0^{s+1} 1 \ldots$ as per case 3.3d. This is identical to the Dyck word constructed from $T' = \mathsf{nextree}(T)$. Therefore, cases 3.3d and 3.4d are equivalent.

$\square$

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

$\square$

3.3 Loopless Implementation

The algorithm described in this section has been implemented in C using a tree node struct that contains fields for each node's parent, leftmost child (firstborn), and right sibling.

The functions `shift_tree_a` and `shift_tree_b` perform the shifts outlined in cases 3.1a and 3.1b respectively; the function `get_initial_tree` generates the first tree in the ordering.

`t` is a parameter equal to the number of non-root nodes in the tree; `visit` is a user-supplied function for visiting a tree.

```
void coolOtree(int t, void (*visit)(node*)){
    node* root = get_initial_tree(t);
    node* o=root->left_child->right_sibling;

    visit(root);
    //o is NULL for the final tree
    while(o){
        if(o->left_child){ //if o has a child
            shift_tree_b(root,o); 3.1b
            o=o->left_child->right_sibling;
        }else{
            if(o->parent == root){
                shift_tree_b(root,o); 3.1b
                o=o->right_sibling;
            }else{
                shift_tree_a(root,o); 3.1a
                o=o->right_sibling;
            }
        }
        visit(root);
    }
}

//implements the shifts specified in in 3.1a
void shift_tree_a(node* root, node* o){
    //get l,p,g
    node* p = o->parent;
    node* l = p->left_child;
    node* g = p->parent;

    //setup p
    //new left_child is l's right sibling
    p->left_child=o->right_sibling;

    //setup l
    //new right sibling is old parent
    //new parent is parent's parent
    l->parent=g;
    l->right_sibling=p;
    g->left_child=l;

    //push o up to root;
    o->parent=root;
    o->right_sibling=root->left_child;
    root->left_child=o;
}

//implements the shift specified in in 3.1b
void shift_tree_b(node* root, node* o){
```
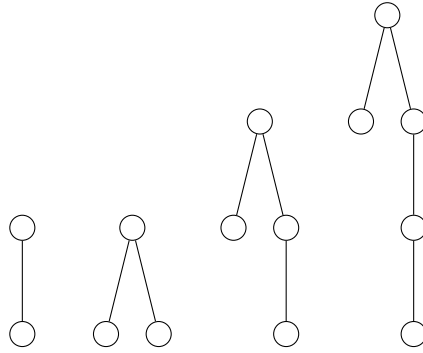
Figure 3.2: The initial trees returned by `get_initial_tree` in `cool0tree` with $t = 1, 2, 3,$ and 4
Note the convenient feature for each tree, O is equal to `root->left_child->right_sibling`

```
//get l
node* l = o->parent->left_child;

//make l o's first child
o->parent->left_child=o;
l->parent=o;
l->right_sibling=o->left_child;
o->left_child=l;
}
```

Algorithm 1 Generate all ordered trees with $t + 1$ nodes

---

    function COOL-ORDERED-TREES(t)
        ▷ Generate initial tree
         TODO
        $O \leftarrow root.lchild$
        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**
                    ▷ Initialize P and L
                    pushchild(O, popchild(P))
                else
                    ▷ Initialize P, L, G
                    pushchild(O, popchild(P))
                    pushchild($root$, popchild(P))
            $O \leftarrow O.rsibling$

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 3.3: Pseudocode and C implementation

Chapter 4

Motzkin, Schröder, and Łukasiewicz Objects

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

In addition to representing balanced parentheses, Dyck paths can be thought of as paths on a cartesian plane. Dyck paths are paths from $(0,0)$ to $(2n,0)$ that use $2n$ steps of either $(1,1)$ (northeast) or $(1,-1)$ (southeast) and never cross below the x axis. In the binary string representation of Dyck words, ones correspond to $(1,1)$ steps and zerores correspond to $(1,-1)$ steps.

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. 4.1 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 graphical representations Notably, Łukasiewicz are a generalization of Motzkin and Schröder paths, as any Motzkin or Schröder path is also a Lukasiewicz path.

The number of Dyck words with n zeroes and n ones are counted by the nth Catalan number. Similarly, the number of Motzkin and Schröder paths of order n are counted by the nth Motzkin and big Schröder number respectively. The number of Lukasiewicz paths of order n are counted by the n 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 $n+1$ nodes. See Fig. 4.2 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.
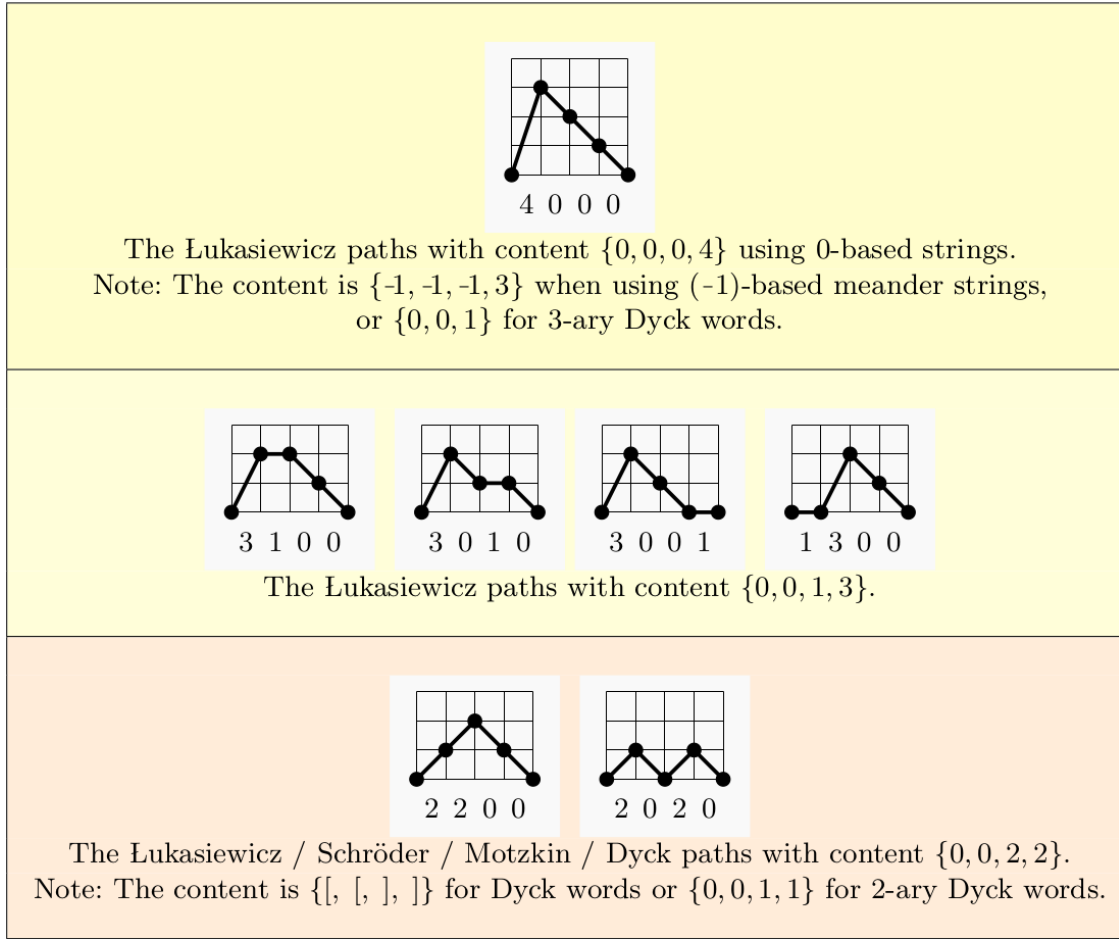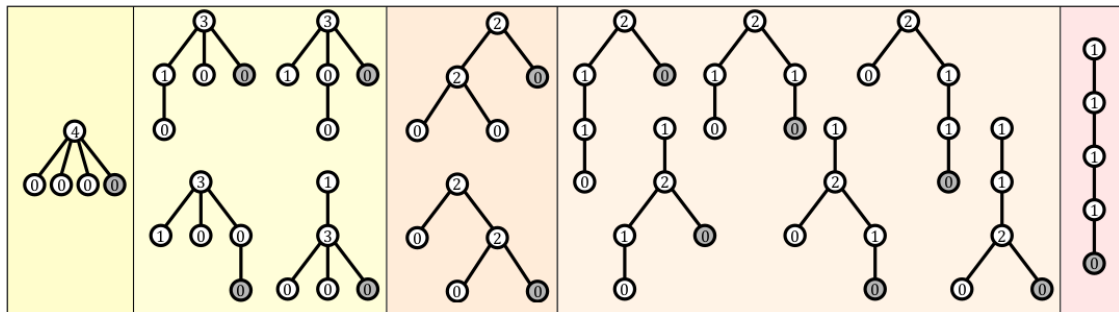
Figure 4.1

Figure 4.2: 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. 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 5

Loopless Motzkin Word Generation

5.0.1   Loopless Motzkin Generation

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. 5.1.

Let m be the length of the longest non-increasing prefix of α.

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

Algorithm 2 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 5.1: Pseudocode algorithm for loopless enumeration of Motzkin words

Chapter 6

Loopless (?) Lukasiewicz Word Generation

6.0.1 Lukasiewicz Path Successor Rule

The successor rule for Lukasiewicz paths is as follows:

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 $\alpha \in \mathcal{L}(S)$.

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

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

In addition to generating Lukasiewicz words, this successor rule also

Chapter 7

Final Remarks

7.1 Summary

7.2 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.

[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.

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

[Wil09] 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.