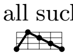
A Shift Gray Code for Fixed-Content Łukasiewicz Words

Paul W. Lapey[1] and Aaron Williams[1][0000−0001−6816−4368]

Williams College, Williamstown MA 01267, USA
{pwl1,aaron.williams}@williams.edu
https://csci.williams.edu/people/faculty/aaron-williams/

Abstract. A Łukasiewicz path of length n is a lattice path from $(0,0)$ to $(n,0)$ that never goes below the x-axis, and which uses steps of the form $(1,i)$ for integers $i \geq -1$. These paths include both Dyck paths ($i \in \{-1,1\}$) and Motzkin paths ($i \in \{-1,0,1\}$). A set of fixed-content Łukasiewicz paths contains all such paths in which the frequency of each step is fixed. For example,  is the only path with one $(1,3)$ step and three $(1,-1)$ steps; equivalently, the only Łukasiewicz word with content $\{-1,-1,-1,3\}$ is $3\ -1\ -1\ -1$ (or 4000 using 0-based values). We contribute a shift Gray code for these fixed-content sets, meaning that consecutive paths differ by moving a single line, and consecutive words differ by moving a single symbol. We also provide a successor rule for generating the next word directly from the current word, as well as loopless array-based algorithms for generating generalized fixed-content Motzkin and Schröder words. Our Gray code generalizes the cool-lex order Gray code for Dyck words.

Keywords: Łukasiewicz path · Łukasiewicz word · Dyck word · Motzkin word · fixed-content · Gray code · cool-lex order.

1 Introduction

When the nodes of an ordered tree are labeled by their number of children, then a preorder traversal gives a Łukasiewicz word. In this paper, we efficiently order and generate Łukasiewicz words. More specifically, we consider sets of fixed-content Łukasiewicz words, which contain strings with the same multiset of symbols (see Figure 1). These sets of strings correspond to ordered trees with the same branching sequence (see Figure 2).

Our first result is a left-shift Gray code for fixed-content Łukasiewicz words, meaning that each string is obtained from the previous by moving one symbol to the left (see Figure 4). There is also a relatively simple successor rule that provides the shift (see (4)) and the resulting order is a cool-lex variant of lexicographic order. Our second result is *loopless* (i.e., worst-case $O(1)$-time per string) array-based implementation for generating the special case of fixed-content Motzkin words. Both the shift Gray code and loopless algorithm generalize previous results for Dyck words, binary trees, and ordered trees [15, 9];

alternate generalizations to k-ary Dyck words [5, 4] and binary bubble languages [13, 23] have also been considered.

To our knowledge, this paper represents the first shift Gray code for fixed-content Łukasiewicz words. Many previous investigations have focused on different orders for related sequences and special cases of these words [2, 3, 8, 11, 20, 21, 24]. For additional background we refer the reader to Knuth's coverage of generating combinatorial objects in Volume 4A of *The Art of Computer Programming* [7], and Mütze's recent update [10] of Savage's classic survey [17].

Section 2 introduces the relevant combinatorial objects, Section 3 provides the successor rule for generating our shift Gray codes, and Section 4 proves that the rule is correct. Section 5 provides our loopless algorithm for fixed-content Motzkin words, with Python code in the Appendix.

2 Background

In this background section, we discuss the combinatorial objects that will be generated in this paper, as well as their history and encodings.

2.1 Lattice Paths: Dyck, Motzin, Schröder, and Łukasiewicz

Lattice paths are well-studied in combinatorics, with books on the subject dating back to the 1970s (see Narayana [12]). In particular, most readers will be familiar with Dyck paths, which are paths from $(0, 0)$ to $(2n, 0)$ using $2n$ steps of the form $(1, 1)$ (north-east) and $(1, -1)$ (south-east), and having the property that the path never goes below the x-axis. These paths can be encoded as *balanced parentheses*, or as integer strings according to several possible encoding schemes.

- North-east steps are 1 and south-east steps are 0. With this encoding, every prefix must have as many 1s as 0s.
- North-east steps are 1 and south-east steps are -1. With this encoding, every prefix must have a non-negative sum.
- North-east steps are 2 and south-east steps are 0. With this encoding, every prefix's sum must be at least as large as its length.

All of these encodings have been referred to as *Dyck words of order n*. We refer to the latter two as the -1-*based encoding* and the 0-*based encoding*, respectively. For example, the five Dyck words of order $n = 3$ are

$$\{[\,]\,[\,]\,[\,], [\,]\,[\,[\,]\,], [\,[\,]\,]\,[\,], [\,[\,]\,[\,]\,], [\,[\,[\,]\,]\,]\} = \{202020, 202200, 220020, 220200, 222000\}$$

when using balanced parentheses and the 0-based encoding, respectively.

Many generalizations of Dyck paths and Dyck words have been studied under the name *generalized Dyck words*. For example, one can consider multiple types of parentheses simultaneously (e.g., '(' with ')' and '[' and ']'), or have longer inequality chains (e.g., every prefix has as many 2s as 1s as 0s).

Another approach is to vary the steps. For example, a *k-ary Dyck path of order n* is a path from $(0, 0)$ to $(kn, 0)$ using kn steps of the form $(1, k - 1)$ and

$(1,-1)$ while never going below the x-axis. The corresponding *k-ary Dyck words* can again be encoded in several ways, and Dyck words are obtained when $k = 2$.

A broader step-based generalization is a *Łukasiewicz path*, which is a path from $(0, 0)$ to $(n, 0)$ that does not go below the x-axis, and which uses steps $(1, i)$ for any integer $i \geq -1$. These paths can be encoded as strings by generalizing either of the last two encodings for Dyck words discussed above.

- -1-*based encoding*: Each $(1, i)$ step is encoded as i, and every prefix must have a non-negative sum.
- 0-*based encoding*: Each $(1, i)$ step is encoded as $i + 1$, and the sum of every prefix must be at least as large as its length.

We prefer the 0-based encoding, and refer to these strings as *Łukasiewicz words of order n*. Figure 1 illustrates all Łukasiewicz paths and words for $n = 4$. Although Łukasiewicz paths include Dyck paths, they differ in their use of n and the term *order*. In particular, the middle row of Figure 1 includes all Dyck words of order $\frac{4}{2} = 2$, since the order of a Dyck word is its number of pairs.

Łukasiewicz paths include Dyck paths when the steps are $(1, i)$ for $i \in \{-1, 1\}$. They also include *Motzkin paths*, where $i \in \{-1, 0, 1\}$. A 0-based encoding is typically used for the corresponding *Motzkin words*, with $\{111, 120, 201, 210\}$ containing the four options when $n = 3$. The closely related *Schröder paths* differ from Motzkin paths in using an east step of $(2, 0)$ rather than $(1, 0)$. For example, the six *Schröder words* of order $n = 2$ are $\{11, 120, 201, 210, 2200, 2020\}$.

The Dyck, Motzkin, and Schröder paths of order n are enumerated by the nth Catalan number, Motzkin number, and big Schröder number, respectively. These sequences are illustrated below for $n \geq 0$ along with their respective entries in the Online Encyclopedia of Integer Sequences (OEIS) [18]:

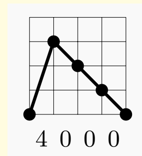
$$\mathcal{C}_n = 1, 1, 2, 5, 14, 42, 132, \ldots \qquad \text{OEISA000108} \tag{1}$$

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

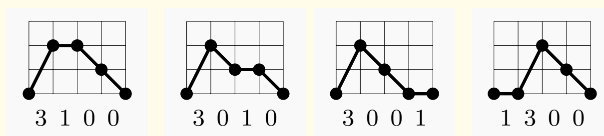
$$\mathcal{S}_n = 1, 2, 6, 22, 90, 394, 1806, \ldots \qquad \text{OEISA006318} \tag{3}$$

The Łukasiewicz paths of order n are enumerated by $\mathcal{C}_{n+2}$. Due to their connections with $\mathcal{C}_n$, $\mathcal{M}_n$, and $\mathcal{S}_n$, these paths are in bijective correspondence with many interesting combinatorial objects, with Stanley's book, *Catalan Numbers*, outlining hundreds of examples [19]. In particular, Łukasiewicz paths have a particularly nice mapping to rooted ordered trees with $n+1$ internal nodes (see Figure 2), and for convenience, each node is labeled by its number of children. These 0-based words have also been referred to as *preorder codewords* [1].

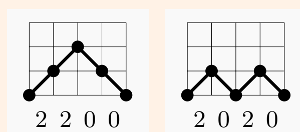
Łukasiewicz paths are named after Jan Łukasiewicz for whom reverse Polish notation is also named. For historical notes on Łukasiewicz's life and mathematics see [6]. When considering Łukasiewicz paths for the first time, it is helpful to note that paths of order n can use steps of maximum slope $(1, n - 1)$, since otherwise there won't be enough $(1, -1)$ steps to return to the x-axis at position $(n, 0)$. This restriction also ensures that there are a finite number of such paths for all n. See [2] for a discussion of more general lattice paths using the Banderier–Flajolet model, including *excursions*, which are paths from $(0, 0)$ to $(n, 0)$ that do not go below the x-axis, and which use steps $(1, i)$ for any integer i.



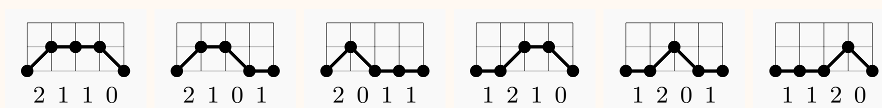
The Łukasiewicz paths with content $\{0,0,0,4\}$ using 0-based strings.
Note: The content is $\{-1,-1,-1,3\}$ when using (-1)-based meander strings, or $\{0,0,1\}$ for 3-ary Dyck words.



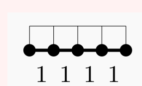
The Łukasiewicz paths with content $\{0,0,1,3\}$.



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.



The Łukasiewicz / Motzkin paths with content $\{0,1,1,2\}$.
Note: The content is $\{d,h,h,u\}$ when referring to *u*p, *d*own, and *h*orizontal moves.



The Łukasiewicz / Motzkin paths with content $\{1,1,1,1\}$.

Fig. 1: All $\mathcal{C}_4 = 14$ Łukasiewicz paths of order 4 are partitioned into rows by their content (i.e., their multiset of slopes). The bottom three rows have all $\mathcal{M}_4 = 9$ Motzkin paths of order 4. The middle row has all $\mathcal{C}_2 = 2$ Dyck paths of order 2. The top row has the $\mathcal{C}_1^3 = 1$ 3-ary Dyck path of order 1. Each row is ordered lexicographically by the path's 0-based string. Other encodings are noted. For example, the second path in the middle row is encoded as 2020 (0-based), $1\ {-1}\ 1\ {-1}$ ((-1)-based), *udud* (moves), [ ] [ ] (Dyck word), or 1010 (2-ary Dyck word). Our main results involve ordering and generating Łukasiewicz words (i.e., the 0-based strings) for a given content (i.e., multiset of symbols). In other words, we focus on the strings listed in the types of rows shown above.

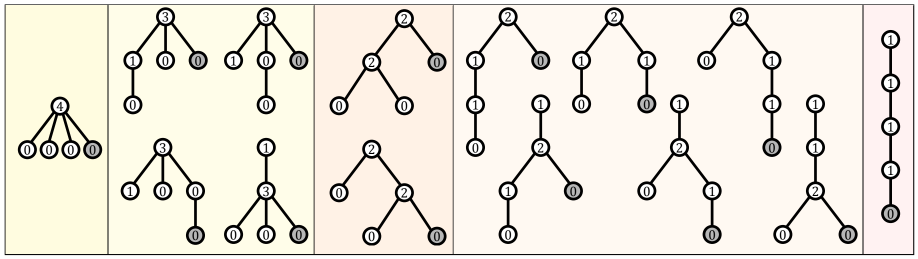


Fig. 2: The $\mathcal{C}_4 = 14$ Łukasiewicz word of order $n = 4$ are in one-to-one correspondence with the rooted ordered trees with $n+1 = 5$ internal nodes. Given a tree, the corresponding word is obtained by recording the number of children of each node in a preorder traversal; the last 0 (from the rightmost leaf) is omitted. For example, the two trees in the middle section correspond to 2200 (top) and 2020 (bottom). The trees are partitioned based on their branching sequence, which corresponds to the content of the associated Łukasiewicz words (see Figure 1).

2.2 Restriction to Fixed-Content

Lattice paths are often restricted in various ways when they are studied. We focus on *content*, which refers to the multiset of symbols used in a word, or equivalently, the multiset of steps used in a path. We use the term *fixed-content* to refer to all Łukasiewicz words, or paths, with the same content. We use $\mathcal{L}(S)$ to denote the set of (0-based) Łukasiewicz words with content S, where S is a multiset of non-negative integers whose sum is equal to its cardinality. For example, the Łukasiewicz paths in Figure 1 are partitioned into fixed-content parts — $\mathcal{L}(\{0,0,0,4\})$; $\mathcal{L}(\{0,0,1,3\})$; $\mathcal{L}(\{0,0,2,2\})$; $\mathcal{L}(\{0,1,1,2\})$; $\mathcal{L}(\{1,1,1,1\})$ — where $\{\}$ or $[]$ denotes multiset content.

The restriction to fixed-content is useful for several reasons. For example, Łukasiewicz paths generalize Dyck paths in the sense that the set of allowed steps is broadened. But it is not true that the set of Łukasiewicz paths of order n generalize the set of Dyck paths of order n; more precisely, they form a superset. On the other hand, fixed-content Łukasiewicz words do generalize Dyck words in this sense. For example, $\{202020, 202200, 220020, 220200, 222000\}$ is both the set of Dyck words of order $n = 3$ (using 0-based encoding), and the Łukasiewicz words with fixed-content $[0,0,0,2,2,2]$. Similarly, fixed-content Łukasiewicz words generalize both fixed-content Motzkin words and fixed-content Schröder words. For example, $\{120, 201, 210\}$ is the set of Motzkin, Schröder, and Łukasiewicz words with content $[0,1,2]$. Note that in this example, the Motzkin and Łukasiewicz words have order $n = 3$, while the Schröder words have order $n = 4$.

The Motzkin and Schröder numbers are partitioned by their content in OEIS A055151 and A088617, respectively. For example, the row **1**, **6**, **2** in the left triangle corresponds to the number of Motzkin objects in the bottom three rows of Figure 1 and the right three columns of Figure 2 (although the order is reversed). The same values appear diagonally in the right triangle due to the differing order of the corresponding Schröder objects. (Due to the greater variety, it is less obvious how to order the analogous quantities for Łukasiewicz words, and the authors did not find a corresponding OEIS sequence.)

A055151	A088617
1	1
1	1 1
1 1	1 3 **2**
1 3	1 **6** 10 5
1 6 2	**1** 10 30 35 14
1 10 10	1 15 70 140 126 42
1 15 30 5	1 21 140 420 630 462 132

Placing a fixed-content restriction on a set of strings can also coincide with a meaningful restriction in corresponding combinatorial objects. For example, restricting Łukasiewicz words to fixed-content corresponds to restricting rooted ordered trees to a specific branching sequence. The *branching sequence* of a rooted tree is the sorted list of the number of children of each node in the tree. For example, the fourth section of Figure 2 shows the ordered trees with branching sequence $0, 0, 1, 1, 2$, which correspond to the Łukasiewicz words with content $\{0, 1, 1, 2\}$ (as one copy of 0 is omitted).

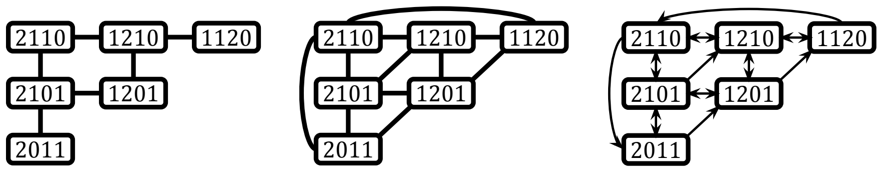
2.3 Gray Codes for Lattice Paths and Strings

In this paper, we are not concerned with counting lattice paths, but in efficiently ordering them. More specifically, we want to create a *minimal-change order*, or *Gray code*, which means sequencing the objects so that each differs from the previous in a specific small way. Our orders are also *cyclic*, in the sense that the last object can be transformed into the first via the same type of small change.

When constructing Gray codes, it is helpful to think about the underlying graph of objects and allowable changes. For example, Figure 3a illustrates the six Łukasiewicz words with content $\{0, 1, 1, 2\}$ as vertices, with edges connecting those that differ by a swap. A *swap*, or *adjacent-transposition*, interchanges two symbols that are immediately next to each other in the string. For example, swapping 20 with 02 changes a peak to a valley in the corresponding lattice path, and it is only valid if the path was above the x-axis at that location prior to the swap. Observe Figure 3a does not have a Hamilton path, so $\mathcal{L}(\{0, 0, 1, 2\})$ does not have a swap Gray code. Thus, we need to broaden our notion of a minimal change in order to create a Gray code for these objects.

One generalization[1] of an adjacent-transposition is a *shift*, in which a single symbol is moved to another position. Figure 3b illustrates the associated graph, and in this case, there is a Hamilton cycle. Thus, there is a cyclic shift Gray code for this set of strings, and one could hope to prove that such a Gray code always exists for fixed-content Łukasiewicz words. We aim slightly higher by considering a more restrictive notion of a minimal-change. A *left-shift* moves a single symbol

[1] Another generalization is a *transposition*, in which two values are interchanged, without the restriction that they must be next to each other in the string.



(a) Swap Graph (b) Shift Graph (c) Left-Shift Graph

Fig. 3: Graphs associated with Gray codes of $\mathcal{L}(S)$ for $S = \{0, 1, 1, 2\}$.

somewhere to the left within a string. More specifically, if $\alpha = a_1 \cdot a_2 \cdots a_n$ is a string and $i < j$, then we let

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

In other words, $\text{left}_\alpha(j, i)$ shifts a_j to the left into position i. Observe that $\text{left}_\alpha(i+1, i)$ is an adjacent-transposition or swap. We also omit α from this notation when the context is clear. The directed graph for $\mathcal{L}(\{0, 0, 1, 2\})$ with left-shifts appears in Figure 3c. This graph has a directed Hamilton cycle, and hence, $\mathcal{L}(\{0, 0, 1, 2\})$ has a cyclic left-shift Gray code. We will establish this result for all sets of fixed-content Łukasiewicz words.

3 Successor Rule

In this section, we provide a *successor rule* that applies a left-shift to a Łukasiewicz word. The rule is given below in (4). In the statement of the rule, we assume that $\alpha = a_1 \cdot a_2 \cdots a_n \in \mathcal{L}(S)$, where S is a multiset whose sum is equal to its cardinality. We also assume that $\rho = a_1 \cdot a_2 \cdots a_m$ is α's *non-increasing prefix*. In other words, $a_1 \geq a_2 \geq \cdots \geq a_m$, and either $m = n$ (i.e., the entire string is non-increasing) or $a_m < a_{m+1}$ (i.e., there is an increase immediately following the prefix). The sum of the symbols in ρ is $\sum \rho = a_1 + a_2 + \cdots + a_m$.

$$\text{next}(\alpha) = \begin{cases} \text{left}(n, 2) & \text{if } m = n & \text{(4a)} \\ \text{left}(m+1, 1) & \text{if } m = n-1 \text{ or } a_m < a_{m+2} \text{ or} & \text{(4b)} \\ & (a_{m+2} = 0 \text{ and } \sum \rho = m) & \\ \text{left}(m+2, 1) & \text{if } a_{m+2} \neq 0 & \text{(4c)} \\ \text{left}(m+2, 2) & \text{otherwise} & \text{(4d)} \end{cases}$$

Figure 4 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 (4a) is not applied. Now consider the conditions in (4b). The second condition is $a_m < a_{m+2}$, which is $a_2 = 0 < 1 = a_4$ for α. Since this is true, $\text{next}(\alpha) = \text{left}(m+1, 1)$ by (4b), 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 4.

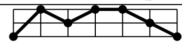
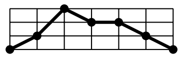
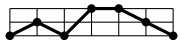
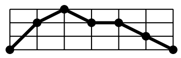
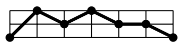
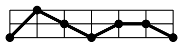
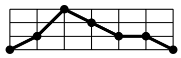
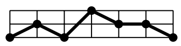
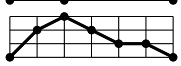
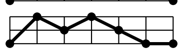
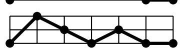
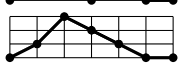
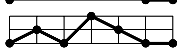
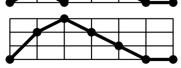
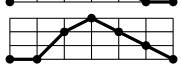
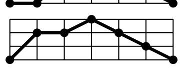
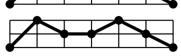
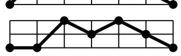
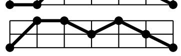
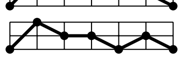
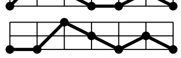
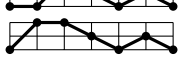
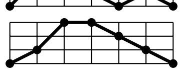
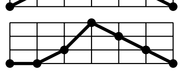
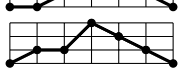
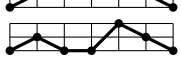
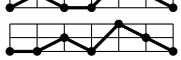
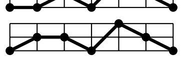
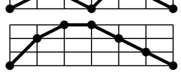
Łukasiewicz path	**Łukasiewicz word**	m	(4)	**shift**	**scut**
	302100	2	(4b)	left$(3,1)$	100
	230100	1	(4d)	left$(3,2)$	100
	203100	2	(4b)	left$(3,1)$	100
	320100	3	(4d)	left$(5,2)$	100
	302010	2	(4d)	left$(4,2)$	10
	300210	3	(4b)	left$(4,1)$	10
	230010	1	(4d)	left$(3,2)$	10
	203010	2	(4b)	left$(3,1)$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

Fig. 4: 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 (4), 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.

3.1 Observations

Note that (4) left-shifts a symbol that is at most two symbols past the non-increasing prefix. Thus, the shifts given by (4) 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 4.

4 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 (4) creates the next string in this order.

4.1 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 [14, 16]. 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 [15] and multiset permutations [22]. 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,

$$\text{tail}(\ell) = t_\ell \cdot t_{\ell-1} \cdots t_2 \cdot t_1, \tag{5}$$

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

$$\text{scut}(s,\ell) = s \cdot \text{tail}(\ell), \tag{6}$$

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 $\mathrm{tail}(n)$ is the only string without a scut, and it is ordered last.

For example, the rightmost column of Figure 4 illustrates this order. More specifically, the scuts appear in the following order:

$$100, 10, 1, 2000, 200, 20, 31000, 3000, 300, \tag{7}$$

with the single string $\mathrm{tail}(n) = 321000$ appearing last. Note that 2, 30 and 3 are absent from (7) 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 $\mathrm{cool}(S')$, where S' is the valid multiset obtained by removing the symbols of the common scut from S. For example, in Figure 4, the strings with scut 1 are ordered according to $\mathrm{cool}(S')$ 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 4, and let $\alpha = \mathrm{tail}(n) = 321000$.

– The first string with a 1-scut is $\mathrm{left}_\alpha(4,2) = 302100$.
– The first string with a 2-scut is $\mathrm{left}_\alpha(3,1) = 132000$.
– The first string with a 3-scut is $\mathrm{left}_\alpha(2,1) = 231000$.

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

4.2 Equivalence

Now we prove that the successor rule (4) correctly provides the next string in $\mathrm{cool}(S)$. This simultaneously proves that (4) is a successor rule for a left-shift Gray code of $\mathcal{L}(S)$, and that $\mathrm{cool}(S)$ is a recursive description of the same.

Theorem 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* $\mathrm{cool}(S)$ *taken circularly (i.e., if α is the last string in* $\mathrm{cool}(S)$*, then β is the first string in* $\mathrm{cool}(S)$*). Then $\beta = \mathrm{left}_\alpha(j,i)$. In other words, the successor rule in* (4) *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 $\text{next}(\alpha) = \text{left}(n, 2)$ by (4a). This gives the first string in $\text{cool}(S)$ with a 1-scut by Remark 1, which is the first string in $\text{cool}(S)$ as expected. This is the only case where (4a) 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 $\text{next}(\alpha) = \text{left}(m+1, 1) = \text{left}(n, 1)$ by (4b). Thus, Remark 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 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 1 and $\text{next}(\alpha) = \text{left}(m+1, 1)$ by (4b).
- 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 1 and $\text{next}(\alpha) = \text{left}(m+1, 1)$ from (4b).
- 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 1 and $\text{next}(\alpha) = \text{left}(m+2, 1)$ by (4c).
- Otherwise, $a_{m+2} = 0$. This is identical to the previous case, except that $a_{m+2} = 0$. Thus, Remark 1 gives $\text{next}(\alpha) = \text{left}(m+2, 2)$ by (4d)

Therefore, (4) gives the next string in the order, which completes the proof.

5 Loopless Algorithm for Fixed-Content Motzkin Words

We now use our Gray code for fixed-content Łukasiewicz words to looplessly generate fixed-content Motzkin words[2]. More specifically, coolMotzkin is an array-based algorithm, and each shift is implemented with a constant number of assignments. Pseudocode is in Figure 5, and Python code is in the Appendix.

The algorithm follows in a similar style to previous array-based algorithms for generating (s,t)-combinations [14, 16], Dyck words [15], and $1/k$-ary Dyck words in cool-lex order [5, 4]. The former two are provided for the sake of comparison in Figure 5 under the names coolCombo and coolDyck, respectively.

A loopless cool-lex algorithm for Łukasiewicz words would require a linked list (as in [22]) since a shift can relocate an arbitrarily number of distinct symbols.

[2] As noted in Section 2.1, these strings are also fixed-content Schröder words.

(a) Combinations

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

(b) Dyck Words

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

(c) Motzkin Words

```
coolMotzkin(s, t)
n ← 2 · s + t
b ← 2^s 1^t 0^s
x ← n − 1
y ← t + s + 1
z ← s + 1
visit(b)
while x < n or b_x < 2 do
  q ← b_{x−1}
  r ← b_x
  if x + 1 ≤ n then
    p ← b_{x+1}
  b_x ← b_{x−1}
  b_y ← b_{y−1}
  b_z ← b_{z−1}
  b_1 ← r
  x ← x + 1
  y ← y + 1
  z ← y + 1
  if p = 0 then
    if z − 2 > x − y then
      b_1 ← 2
      b_2 ← 0
      b_x ← r
      x ← 3
      y ← 2
      z ← 2
    else
      x ← x + 1
  else if x ≤ n and q ≥ b_x then
    b_x ← 2
    b_{x−1} ← 1
    b_1 ← 1
    z ← 1
  if b_2 > b_1 then
    z ← 1
    y ← 2
    x ← 2
  visit(b)
```

Fig. 5: Algorithms for generating (a) (s,t)-combinations, (b) Dyck words, and (c) fixed-content Motzkin words in cool-lex order. The algorithms are loopless and store the current string in array $b = b_1 b_2 \cdots b_n$ (i.e., 1-based indexing). The parameters $s \geq 2$ and $t \geq 2$ give the number of 0s (and 2s) and 1s, respectively. Variables z, y, and x given the index after the 2s, 1s, and 0s in the non-increasing prefix, respectively. (Their initial values are exceptions to this pattern, and are set to make the first iteration work correctly.) The start of the **while** loop shifts the first increasing symbol to the left (i.e., (4b) in coolMotzkin) and the **if** statements identify when this is not the correct shift, and adjust b accordingly. Also, coolMotzkin uses q, r, p to save the symbols around the first increase.

References

1. Balakirsky, V.B.: A new coding algorithm for trees. The Computer Journal **45**(2), 237–242 (2002)
2. Banderier, C., Wallner, M.: The kernel method for lattice paths below a line of rational slope. In: Lattice path combinatorics and applications. Springer (2019)
3. Dershowitz, N., Zaks, S.: Enumerations of ordered trees. Discrete Mathematics **31**(1), 9–28 (1980)
4. Durocher, S., Li, P.C., Mondal, D., Ruskey, F., Williams, A.: Cool-lex order and k-ary Catalan structures. Journal of Discrete Algorithms **16**, 287–307 (2012)
5. Durocher, S., Li, P.C., Mondal, D., Williams, A.: Ranking and loopless generation of k-ary Dyck words in cool-lex order. In: International Workshop on Combinatorial Algorithms. pp. 182–194. Springer (2011)
6. Hodgson, J.: Rediscovered: the Jan Łukasiewicz Papers. https://rylandscollections.com/2018/05/16/rediscovered-the-jan-lukasiewicz-papers
7. Knuth, D.E.: Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees–History of Combinatorial Generation. Addison-Wesley (2013)
8. Korsh, J.F., LaFollette, P.: Loopless generation of trees with specified degrees. The Computer Journal **45**(3), 364–372 (2002)
9. Lapey, P.W., Williams, A.: Pop & push: Ordered tree iteration in O(1)-time (2022), manuscript submitted for publication
10. Mütze, T.: Combinatorial Gray codes-an updated survey. arXiv preprint arXiv:2202.01280 (2022)
11. Nakano, S.i.: Listing all trees with specified degree sequence (acceleration and visualization of computation for enumeration problems). RIMS Kôkyûroku Bessatsu **1644**, 55–62 (2009)
12. Narayana, T.V.: Lattice Path Combinatorics with Statistical Applications; Mathematical Expositions 23. University of Toronto Press (1979)
13. Ruskey, F., Sawada, J., Williams, A.: Binary bubble languages and cool-lex order. Journal of Combinatorial Theory, Series A **119**(1), 155–169 (2012)
14. Ruskey, F., Williams, A.: Generating combinations by prefix shifts. In: International Computing and Combinatorics Conference. pp. 570–576. Springer (2005)
15. Ruskey, F., Williams, A.: Generating balanced parentheses and binary trees by prefix shifts. In: Proceedings of the fourteenth symposium on computing: the Australasian theory-Volume 77. pp. 107–115. Citeseer (2008)
16. Ruskey, F., Williams, A.: The coolest way to generate combinations. Discrete Mathematics **309**(17), 5305–5320 (2009)
17. Savage, C.: A survey of combinatorial Gray codes. SIAM review **39**(4), 605–629 (1997)
18. Sloane, N.J.A., The OEIS Foundation Inc.: The on-line encyclopedia of integer sequences (2020), https://oeis.org/
19. Stanley, R.P.: Catalan numbers. Cambridge University Press (2015)
20. Van Baronaigien, D.R.: A loopless algorithm for generating binary tree sequences. Information Processing Letters **39**(4), 189–194 (1991)
21. Wallner, M.: Combinatorics of lattice paths and tree-like structures. Ph.D. thesis, Wien (2016)
22. Williams, A.: 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. pp. 987–996. SIAM (2009)

23. Williams, A.M.: Shift Gray codes. Ph.D. thesis, University of Victoria (2009)
24. Zaks, S., Richards, D.: Generating trees and other combinatorial objects lexicographically. SIAM Journal on Computing **8**(1), 73–81 (1979)

Appendix: Python Code

Python3 functions for generating the cool-lex order of (s,t)-combinations, Dyck words of order t, and fixed-content Motzkin words with s copies of 0 and 2 and t copies of 1, are found in Figure 6[3]. The first two are found in [14, 16] and [15], respectively, and the latter is new to this article. To simulate the 1-based indexing used in Figure 5, we store array `b` in a list and ignore its first entry `b[0]` . Lists are implemented as arrays in CPython, so each read and write is a worst-case $O(1)$-time operation. Hence, the implementations are loopless.

```
def coolCombo(t,s):
..n = s+t
..b = [-1]+[1]*t+[0]*s
..x = t
..y = t
..print(*b[1:],sep="")
..while x < n:
....b[x] = 0
....b[y] = 1
....x += 1
....y += 1
....if b[x] == 0:
......b[x] = 1
......b[1] = 0
......if y > 2:
........x = 2
......y = 1
....print(*b[1:],sep="")
```

```
def coolDyck(t):
..n = 2*t
..b = [-1]+[1]*t+[0]*t
..x = t
..y = t
..print(*b[1:],sep="")
..while x < n-1:
....b[x] = 0
....b[y] = 1
....x += 1
....y += 1
....if b[x] == 0:
......if x >= 2*y - 2:
........x += 1
......else:
........b[x] = 1
........b[2] = 0
........x = 3
........y = 2
....print(*b[1:],sep="")
```

```
def coolMotzkin(t,s):
..n = 2*s + t
..b = [-1]+[2]*s+[1]*t+[0]*s
..x = n-1
..y = t+s+1
..z = s+1
..print(*b[1:],sep="")
..while x < n-1 or b[x] < 2:
....q = b[x-1]
....r = b[x]
....if x + 1 <= n:
......p = b[x+1]
....b[x] = b[x-1]
....b[y] = b[y-1]
....b[z] = b[z-1]
....b[1] = r
....y += 1
....z += 1
....x += 1
....if p == 0:
......if z-2 > (x-y):
........b[1] = 2
........b[2] = 0
........b[x] = r
........z=2
........y=2
........x=3
......else:
........x+=1
....elif x <= n and q >= b[x]:
......b[x] = 2
......b[x-1] = 1
......b[1] = 1
......z = 1
....if b[2] > b[1]:
......z = 1
......y = 2
......x = 2
....print(*b[1:],sep="")
```

Fig. 6: Loopless generation of the cool-lex shift Gray codes of (s,t)-combinations, Dyck words, and fixed-content Motzkin words in Python 3. Each shift is achieved using a constant number of assignments to the list `b`.

[3] The leading spaces have been replaced with periods to ensure that the code can be reliably copy-and-pasted from digital versions of this document.